12-2005

Obtaining High Precision Results from Low Precision Hardware

Adam Donald Graham
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Adam Donald Graham entitled "Obtaining High Precision Results from Low Precision Hardware." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael D. Vose, Major Professor

We have read this thesis and recommend its acceptance:

Bruce MacLennan, David Straight

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:
I am submitting herewith a thesis written by Adam Donald Graham entitled "Obtaining High Precision Results from Low Precision Hardware". I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael D. Vose
Major Professor

We have read this thesis
and recommend its acceptance:

Bruce MacLennan

David Straight

Accepted for the Council:

Anne Mayhew
Vice Chancellor and Dean of
Graduate Studies

(Original signatures are on file with official student records.)

Obtaining High Precision Results from Low Precision Hardware

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Adam Donald Graham
December 2005

Dedication

This thesis is dedicated to Bruce Graham, my father and friend, you'll never be forgotten. Also to my mother, Cynthia Graham, and brothers Lucas and Chris whose love and support were invaluable during the last few years.

Abstract

This document describes an attempt at acheiving high precision matrix multiplication results from the Lenslet EnLight256 Optical Signal Processor (OSP), which on its own can only produce results which are hardware limited to 8-bit signed integers. Due to it's low precision, it's has only limited applicability to real world problems, and if higher precision results were possible from the machine it could be used for more applications. A C library is developed for this thesis to allow high-precision results from the EnLight256. The library is described and results are given. Finally an implementation of the Jacobi Method on the EnLight256 is given as an example of the library being used in a real world scenario.

Contents

List of Figures

Chapter 1

Introduction

The Lenslet EnLight256 is an 8-bit Optical Signal Processor capable of performing matrix/vector multiplies in a single cycle. This makes it a very attractive piece of hardware for speeding up certain matrix/vector multiply intensive tasks. Unfortunately its limited precision of 8 bits cripples the range of possible uses, as an effective precision range of $\{-128, \ldots, 127\}$ limits its applicability. It is rumored that a 12-bit version of the device could be built, but that a simulator for that device was not available. Also, the maximum size on the machine for any matrix is 256×256, and the largest possible vector is 256 entries long. The methods outlined in this document could with little effort be modified to take advantage of the extra 4 bits. Since actual hardware was not available to the author this thesis used a simulator for the EnLight256 developed by Dr. Michael D. Vose, and documented in [Vose, 2004]. Also a theoretical treatment of obtaining high-precision results from low-precision hardware can be found in [Grimmell, 2005].

This thesis attempts to extend the EnLight256's precision while minimizing the loss in overall calculation speed that such an attempt may potentially cause. A successful attempt would allow the EnLight256 to be applied to a broader range of applications than it is natively capable of. At the same time it is hoped a "tool set" of functions can be developed to allow programmers in general to use the EnLight256 as a high-precision device. All programming was done in the C language on a variety of machines and architectures, and compiled using gcc 4.0.

In essence, the problem consists of breaking a high precision matrix and vector that are to be multiplied into lower precision parts small enough for the EnLight256 to multiply on its own. This needs to be done without any kind of data loss, and while ensuring the final answer remains correct. The methods used to achieve the above are compartmentalized into functions to allow them to be used for general purposes.

1.1 Document Organization

This document is presented in the the form of a series of programs and functions that were written to explore and work through the problems that arose in attempting to extend the precision of the EnLight256. The programs in the second chapter are generally built incrementally one on top of the last, and each program includes improvements over the

methods used in the previous program and/or some new functionality. In each section a description of what the aims of the program were is given. Following that, particular problems that arose during the design of the program along with the attempted solutions to those problems are described. Functions applicable to the program are then described which implement certain aspects of the needed procedure to raise the functional precision level of the device, and each function from the second chapter is one more tool in the kit for extending the precision of the EnLight256. As a matter of course all direct references to the C code written for the project and the names of the files that contain it are in `this font`, and references to mathematical entities are presented in $thisfont$.

After the methods have been determined and a tool set developed in the form of the functions described in the program chapter, the tools are used in a new set of programs to test their applicability in a series of problems. In the chapters are inclued a brief description of the problem and the effects of using the developed tool set to solve them.

Finally, the appendix contains the programs themselves for the reader to reference.

Chapter 2

Programs

During the course of the thesis programs were written to incrementally tackle difficulties that arose with attempting to achieve high precision results from the EnLight256 and to develop a general tool set. Following are descriptions of these programs and explanations of the difficulties they were designed to overcome, and the method used to do so for each.

2.1 Making the Simulator Name Space Safe

The first task to accomplish was to adjust the EnLight256 simulator to reduce the potential for name space clashes (the occurrence of the simulator and the program calling it using the same variables names and potentially causing undefined behavior). This was achieved by adding to all non-interface related (thus "invisible" to the user) function names and variables the prefix of `__EL256_`. This was done to help ensure the simulator's safe use over a variety of systems and with as much user code as possible.

2.2 onebit.c

The main goal of this program was to multiply a randomly generated matrix composed of one-bit numbers by a randomly generated vector composed of one-bit numbers on the simulated hardware using the EnLight256's `APL_VMM` (Vector Matrix Multiply) command.

This program, the code for which can be found in appendix A, was essentially written as practice, but even given this program's limited scope issues needed to be addressed with respect to "hard-limiting". "Hard-limiting" refers to the rounding off of any result outside the hardware's range of precision of $\{-128, \ldots, 127\}$ to the closest value within the range of its precision by the hardware. I.e. 129 would be rounded to 127, and -129 would be rounded to -128 . Firstly it was required that the arguments passed to the `APL_VMM` command be adjusted to compensate for the 15-bit right-shift that the hardware applies to the results of the command, as described in [Vose, 2004]. Thus to get the correct result from the command the program needs to overcome the hardware shift by multiplying the input by a total of 2^{15}, which is the equivalent of a left-shift by 15 bits. Given an $n \times n$ matrix m and vector v of length n, both filled with 8-bit `signed char` entries of which only the lowest order bit position is used, we can safely shift each entry in m and v left by

$8-1=7$ bits maximum without pushing the used bit out of the register. Doing this is the equivalent of multiplying each entry in m and v by 2^7. Since m and v are being multiplied the applied shifts have the net result of $2^7 \cdot 2^7 = 2^{14}$, or a 14-bit left-shift to the final result. Unfortunately we need a shift of at least 15 bits to overcome the shift applied by the hardware to the result. Fortunately the command `APL_VMM` allows the user to specify a left-shift of $s \in \{0, \ldots, 6\}$ to apply to the result of the multiplication. Given that we want to compensate for a shift of 2^{15} and have a shift of 2^{14} already, setting $s = 1$ will yield $2^{14} \cdot 2^1 = 2^{15}$, and since $2^{15} \cdot 2^{-15} = 1$ the hardware applied shift is neutralized and the result from `APL_VMM` will be correct. The technique for determining shifts is generalized for multiple-bit matrices and vectors in the next section for the program `1-3bit.c`.

Secondly, even when using only one-bit entries to fill the matrix and vector, consequences of the limited precision of the EnLight256 arise. Consider multiplying a $n \times n$ matrix m by a length n vector v, where $n = 256$ (the maximum number of entries allowed), every entry in m and v has type `signed char`, and every entry in m and v set equal to 1. Multiplying row j of m by v will yield a result of $2^8 = 256$ which is greater than the max precision of 127 allowed by the machine. Thus the result will be hard-limited to 127. So even in the lowest precision case of one-bit entries there exists a need to break the matrix up into smaller parts in order to keep full precision. To effect this the program was modified to create two matrices, m_1 and m_2, with each matrix receiving one half of the entries of the original matrix m. This has the effect of limiting the maximum result to $2^7 = 128$. However since 128 is greater than the maximum precision of 127 there can still be precision loss. This problem is generalized and solved in the next section for the program `1-3bit.c`.

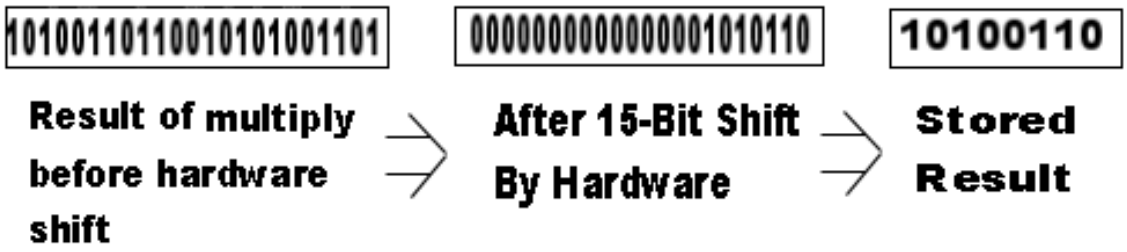
2.3 1-3bit.c

This program was written as an extension to `onebit.c` to increase the number of bits allowed in the entries in the matrix m and the vector v to 3 bits each, where each 3-bit entry in m and v is stored as an 8-bit `signed char`. Also, it implements generalized versions of the method used to overcome the 15-bit hardware right-shift, and the method used to overcome the hard-limiting, encountered during the writing of `onebit.c`. Namely `1-3bit.c` determines the amount by which to shift m and v, the amount of shift s to apply to the `APL_VMM` command and how to "partition" m to ensure the results of the multiply are within the range of the hardware's precision limits. The code for `onebit.c` can be found in appendix B.

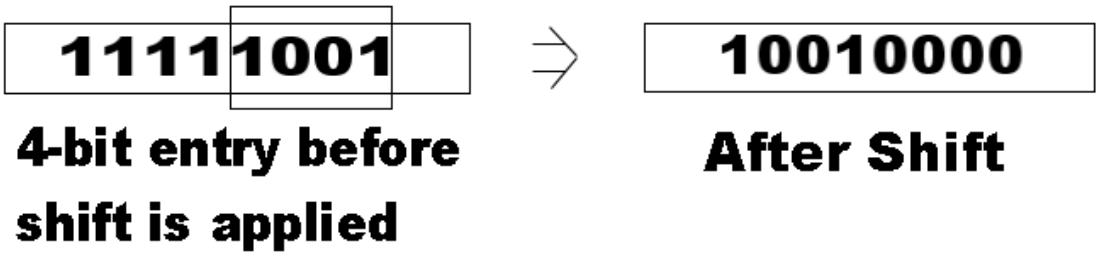
2.3.1 Determining Shifts

The first problem to overcome was to determine how much to shift the entries in both m and v by, and how to use those values to determine s (the shift supplied to the `APL_VMM` command) to overcome the 15-bit right-shift applied by the hardware as shown in figure 2.1(a).

Firstly, it should be noted that $s \in \{0, \ldots, 6\}$, meaning that there are limits on the amount of shift that can be applied to m and v. For instance, assume m and v are filled with 4-bit entries (larger than we will be dealing with in `1-3bit.c`, but useful for



(a)



(b)

Figure 2.1: Shifting Examples. (a) The shift applied by the EnLight256 to all matrix/vector multiply results. (b) An example shift of a 4-bit entry. Note the sign is maintained.

illustration). Given that these entries are stored as 8-bit `signed chars`, there is room to apply a left shift of 4 bits to the entries of both m and v without losing any information. That gives a total shift of 8, meaning a shift s of $15 - 8 = 7$ would be needed to overcome the hardware imposed shift on the results. However $7 \notin \{0, \ldots, 6\}$. Therefore, multiplying a 4-bit m by 4-bit v is not natively possible using the `APL_VMM` command.

From this one can determine the maximum shift applicable to m and v such that $0 \leq s \leq 6$ and that the `APL_VMM` command will be successful (i.e. the 15-bit hardware shift to the result of the multiply will be nullified). Let m_b and v_b represent the maximum number of bits used for the entries, including the sign bit, in m and v respectively and let s_m be the shift applied to m, s_v the shift applied to v. Since the entries are stored as `signed chars`, we can determine the maximum applicable shift without data loss for m as

$$s_m = 8 - m_b \tag{2.1}$$

and for v as

$$s_v = 8 - v_b \tag{2.2}$$

So long as $s_m, s_v \geq 0$, the sign of entries will be maintained and all used bits will remain intact, as shown in figure 2.1(b). Given that we are trying to overcome a shift of 15, s can be determined by

$$s = 15 - s_m - s_v \tag{2.3}$$

In `1-3bit.c` $m_b = v_b$, thus the code need only calculate s_m and

$$s = 15 - (2 \cdot s_m) \tag{2.4}$$

In `1-3bit.c` s is represented by `vmm_shift`, and s_m by `numbits`. In later programs m_b and v_b are allowed to differ. Thus the more general version of the shift determination described by equation 2.3 will be implemented. By this method the shifts needed to do a successful matrix/vector multiply on the EnLight256 using the `APL_VMM` command can be determined at run time.

From this we can also determine the maximum total number bits that the entries of m and v can contain, in order to avoid hard-limiting causing precision loss on the multiplied results. If the 15-bit hardware right-shift is not to cause data loss, then

$$s_m + s_v + 6 \geq 15 \tag{2.5}$$

where the 6 represents the maximum value of s. Equation 2.5 can be stated differently as

$$(8 - m_b) + (8 - v_b) + 6 \geq 15 \Rightarrow 22 - (m_b + v_b) \geq 15 \Rightarrow m_b + v_b \leq 7 \tag{2.6}$$

Given that both m_b and v_b must be at least 1 we get $(m_b + v_b) \in \{2, \ldots, 7\}$. For `1-3bit.c` we deal with at most $m_b = v_b = 3$ and thus 6 bits total. Hence we will be able to determine appropriate shifts that will generate valid results since $6 \in \{2, \ldots, 7\}$. Note however that the maximum number of bits that can be dealt with using the `APL_VMM` command as determined by equation 2.6 is 7.

2.3.2 Partitioning Problem

The second problem to overcome deals with the necessity to break up the $n \times n$ matrix m into "partitions", i.e. into separate groups of columns that will be multiplied independently, to ensure no result from the `APL_VMM` command will be outside the range of the hardware's precision of $\{-128, \ldots, 127\}$, as shown in figure 2.2.

The first step is to determine the maximum number 2^e of entries per row (and that number will be the maximum number of columns in the partitions), that can comprise a single partition without the EnLight256's hard-limiting causing any loss of precision. Let matrix m have m_b-bit entries and vector v have v_b-bit entries. Given that the entries of m and v are signed, they will have a maximum absolute magnitude of 2^{m_b-1} and 2^{v_b-1} respectively. The largest result of any row in m multiplied by v can be represented as

$$2^e \cdot 2^{m_b-1} \cdot 2^{v_b-1} < 2^7 \tag{2.7}$$

where the right hand side of the inequality represents the need to keep the result smaller than the EnLight256's hard-limit. This can be simplified to

$$e + m_b - 1 + v_b - 1 < 7 \Rightarrow \tag{2.8}$$

$$e < 9 - m_b - v_b \tag{2.9}$$

In `1-3bit.c`, the largest possible value for an entry in m or v would be attained with $m_b = v_b = 3$, and plugging these values into inequality 2.9 yields

$$e < 9 - 3 - 3 \Rightarrow e < 3 \Rightarrow 2^e < 2^3 \tag{2.10}$$

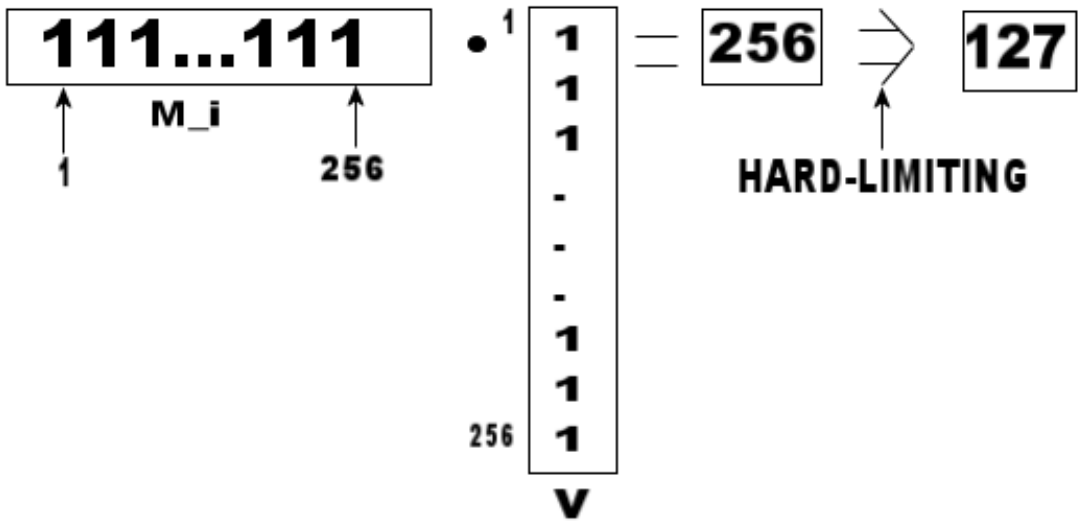
and it becomes clear that a partitioning strategy exists that will work for the program. The required number of partitions p, where p is an integer > 0, can be obtained as follows. The largest value of e satisfying inequality 2.9 is given by

$$2^e = 2^{9-m_b-v_b} - 1 \tag{2.11}$$

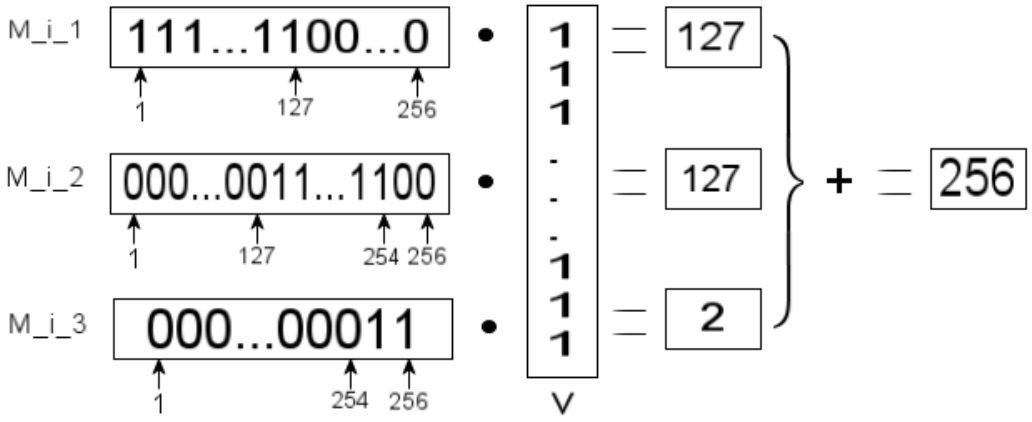
The required number of partitions is then obtained by dividing the number n of entries per row (in m) by the result of equation 2.11 and rounding up. Thus

$$p = \lceil \frac{n}{2^{9-m_b-v_b} - 1} \rceil \tag{2.12}$$

The implementation of partitioning in `1-3bit.c` differs from the method that was used in `onebit.c`. There a new matrix was created for each needed partition, with all of the unused columns zeroed out to implement the partitioning. The method used to implement partitioning in `1-3bit.c` instead uses vector masks bitwise ANDed with v to create v_i, where the subscript $i \in \{0, \ldots, p-1\}$ corresponds to the partition to be multiplied. To implement this p vector masks are created with all of the entries corresponding to columns not included in the partition zeroed out and the entries corresponding to columns contained within the partition set to -1 (-1 was chosen due to its bitwise hardware representation being all 1's). Once v and m are shifted and loaded into the EnLight256, the mask i



(a)



(b)

Figure 2.2: Basic Matrix Partitioning. (a) An example of hard-limiting affecting the outcome of a matrix/vector multiply on the EnLight256. Shown is the i^{th} row in M being multiplied by the vector v, both of which consist of 256 entries set to 1. (b) An example of the basic matrix partitioning scheme. Here the original matrix M has been broken up into 3 matrix partitions. Thus the same row as in (a) is being multiplied, however it has been broken into three partitions that will be multiplied separately and summed externally to achieve the correct result.

corresponding to the current partition being multiplied is also loaded into the EnLight256 and the `APL_AND` command is used to AND v with vector mask i to create v_i. After every partition is multiplied using the `APL_VMM` command the results are summed to create a total result. This method has the advantage of reducing I/O to the hardware since fewer matrices are being loaded (regardless of the actual number of used entries in any matrix or vector the load size will be of either 256x256 or 256 entries respectively). It further has the advantage lowering the overhead from initializing entire matrices to only initializing vector masks and ANDing them on the hardware with v.

2.4 4-3maxmul.c

`4-3maxmul.c`, code for which can be found in appendix C, was designed as an incremental improvement over `1-3bit.c`, mainly to implement the results of what was learned from the work done on `1-3bit.c` and compartmentalize the separate functions for ease of use in later applications. The main difference in the functioning of this program is that it will allow the number of bits used for the entries in the matrix and vector to differ, and allow the total maximum possible number of bits, 7, as determined by the shifting equation 2.6. Also implemented was a new strategy for implementing the vector masks described in 2.4.2, and the summation of the partial results from multiplies of partitions of the matrix was moved onto the EnLight256 as described in section 2.4.3. Following are descriptions of the functions that were created and any changes that were made to their implementations since `1-3bit.c`.

2.4.1 mvshift

Function `mvshift` will determine the results of equation 2.3 and apply, using integers `mbits` and `vbits` representing m_b and v_b, the appropriate shift to the entries of both vector v and matrix m. `mvshift` then stores the shifted results in the vector `sv` and the matrix `sm` respectively, and returns the amount of shift s to be applied to the `APL_VMM` command. NOTE: This function will be compartmentalized further into three sub-functions which are described in 2.6.

2.4.2 genvmask

In `4-3maxmul.c`, the partition strategy described in 2.3.2 in equations 2.11 and 2.12 has been modified to reduce the number of vector loads, and thus costly I/O to the EnLight256, and make further use of the capabilities of the hardware. The method used in `4-3maxmul.c` instead only has one vector mask, `v_mask` of type `vec_type`, being created using the number of bits `mbits` used in the entries in the matrix and the number of bits `vbits` to be used in the entries of the vector.

The entries of the vector mask v_m are filled with either -1 or 0. The value is determined by the following: For integer i, where $0 < i \leq 256$, the i^{th} entry of v_m is -1 if $i \equiv 0 \mod p$. If $i \not\equiv 0 \mod p$, then the i^{th} entry of v_m is 0. Recall that when v_m is ANDed with the vector v to be multiplied only those entries in v that correspond to a -1 in the mask participate in the result, and so only the corresponding columns in the matrix m

contribute to the multiply (the rest are nullified by a multiply by 0). Once a multiply has occurred on a partition of the matrix (using the `APL_VMM` command) the entries of v_m are shifted using the `APL_SHFT_U` command to allow the next partition to be multiplied. Once all columns in m have been multiplied in this fashion the multiply is complete.

2.4.3 HP_MUL

The function `HP_MUL` will perform the actual matrix/vector multiply. The first three arguments to the function do not take the data structures themselves but the registers in which they reside on the EnLight256, meaning they must be loaded into the hardware before the function is called. Keeping in accordance with [Vose, 2004], S_i, L_i, M_i will represent the i^{th} 8-bit short vector, 16-bit long vector and 8-bit Matrix registers respectively on the EnLight256. Both vector registers are 256 entries long and each matrix register is 256×256 entries large. `int m` refers to the matrix register, M_m, in which the shifted matrix being multiplied resides. `int v` signifies the short vector register containing the shifted vector, S_v, and `int mask_reg` the short vector register containing the vector mask, S_{mask_reg}, as produced by the function `genvmask` as described in 2.4.2. `int sum_reg` specifies the long vector register, L_{sum_reg} that will be used to temporarily store the partial sum of the partition multiplies. `mbits` and `vbits` represent the number of bits used in the entries of the matrix and vector, and finally `vmm_shift` is the shift that will be supplied as the last argument to the `APL_VMM` command.

The multiplication method used in `HP_MUL` is illustrated in pseudo code form using registers in figure 2.3. The partition method implemented is described in 2.4.2 and summation of the partial results after each partition is multiplied has been moved onto the EnLight256 to make use of the speed of the hardware (as opposed to being done by the calling function). The internal summation uses a 16-bit long vector register, L_{sum_reg}, and the EnLight256's 16-bit vector add command `APL_VADDM`. Since the program will be using at most 7 bits for entries of the matrix and vector (as determined in equation 2.6) and there can be at most $256 = 2^8$ entries in the matrix the largest possible result from the summation of the parts is of size $2^7 \cdot 2^8 = 2^{15}$ and thus will fit safely inside the 16-bit vector register. Unfortunately, the 16-bit register cannot be used with the `APL_VMM` command itself and hence the need for the partitioning of the matrix.

2.5 8bitmm.c

`8bitmm.c` was written using the functions created in `4-3maxmul.c` and was designed to extend the number of bits that can be multiplied to a total of 16, with a limit of 8 bits on the size of the entries in both the matrix and vector to be multiplied. Like the previous programs, `8bitmm.c` will generate and multiply an $n \times n$ matrix m by a vector v of length n, where $0 < n \leq 256$. In order to accommodate having more than a total of 7 bits (the previous limit determined by inequality 2.6) it was necessary to decompose the matrix and vector into parts small enough to be multiplied using the functions written for `4-3maxmul.c` while at the same time ensuring that no data would be lost. Following is a description of the method used to decompose m and v and descriptions of the functions that implement the method in the code for `8bitmm.c`, which can be found in appendix D.

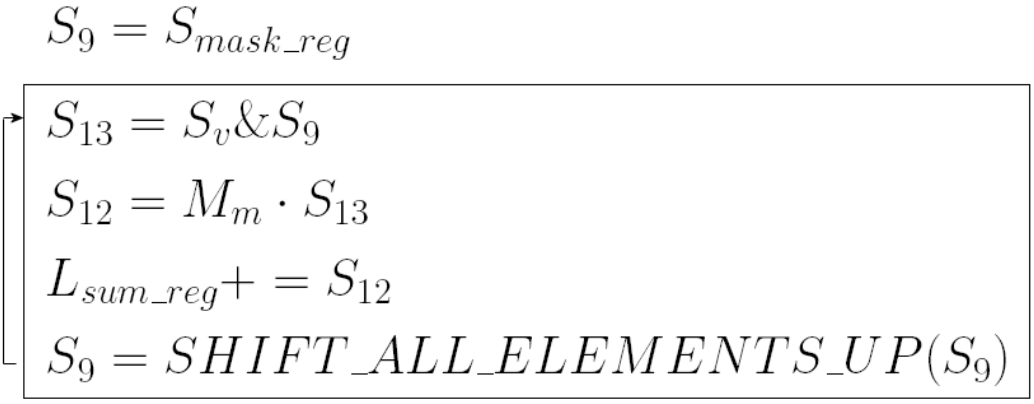


Figure 2.3: HP_MUL EnLight256 Pseudo-Code. The pseudo-code for the HP_MUL command with regards to its usage of EnLight256 registers. The arrow represents a loop back to the first command inside the box which occurs until every partition has been multiplied. S_i is the i^{th} short 8-bit register, L_i is the i^{th} long 16-bit register, and M_i is the i^{th} matrix register, described fully in [Vose, 2004]. SHIFT_ALL_ELEMENTS_UP corresponds to the APL_SHFT_U EnLight256 command.

2.5.1 Decomposition Method

In order for a matrix m and vector v composed of entries greater than 7 bits total to be multiplied using the `HP_MUL` function it is necessary for both to be broken up into "groups" small enough to be used by the `HP_MUL` function as described in section 2.4.3. A "group" consists of a decomposed part of either m or v. Each entry in the group consists of m_d bits for matrix groups and v_d bits for vector groups, with the most significant bit of the group entry representing the sign of the original entry and all bits of lesser significance representing the value of the decomposed entry. Thus for m with m_b-bit entries there will need to be

$$k = \lceil \frac{m_b}{m_d - 1} \rceil \tag{2.13}$$

matrix groups, and for v with v_b-bit entries there will be

$$l = \lceil \frac{v_b}{v_d - 1} \rceil \tag{2.14}$$

vector groups. As shown in figure 2.4, l and k are positive integers with $2 \leq (m_d + v_d) \leq 7$ (from inequality 2.6) and $m_d, v_d > 0$. Note that the sign (most significant) bit of the original entries are considered as information bits when the number of groups needed for decomposition of m and v is decided, hence the numerators of equations 2.13 and 2.14 do not have 1 subtracted from them. This is done to represent the fact that the highest significant bit of the entries on the EnLight256 represent both the sign of the entry and also hold information on the value of the entry when it is the only bit "turned on," in which case the value represented is -128.

However if it can be guaranteed that no entry will have the value -128, 1 can be subtracted from the numerators of equations 2.13 and 2.14 with a possible reduction in the number of required groups (and thus potentially less I/O since one less matrix or vector group will need to be loaded.) For integers i and j, the decomposition of m and v (into groups having values $m(0), \ldots, m(k-1)$ and $v(0), \ldots, v(l-1)$ respectively) can be described by

$$m = \sum_{i=0}^{k-1} m(i) \cdot 2^{i(m_d - 1)} \tag{2.15}$$

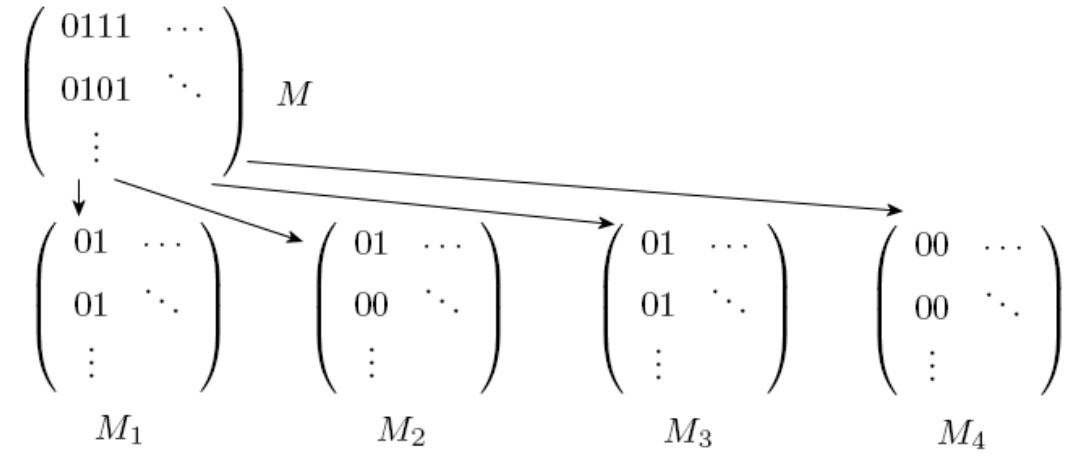
and

$$v = \sum_{j=0}^{l-1} v(j) \cdot 2^{j(v_d - 1)} \tag{2.16}$$

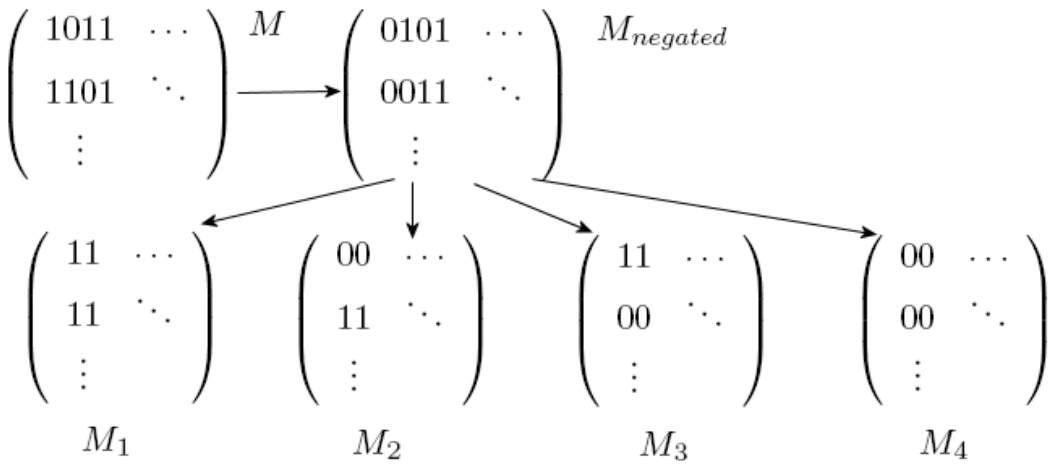
respectively, and $m \cdot v = v_{sum}$ cab be determined by

$$v_{sum} = \sum_{i=0}^{k-1} \sum_{j=0}^{l-1} m(i) \cdot v(j) \cdot 2^{i(m_d - 1) + j(v_d - 1)} \tag{2.17}$$

where $m_i \cdot v_j$ is represented in the code by the multiplication method used in `4-3maxmul.c` and described in 2.4, and the multiplication by $2^{i(m_d-1)+j(v_d-1)}$ is achieved by a shift of $i(m_d - 1) + j(v_d - 1)$ to each element in the resulting vector before it is added to v_{sum} in the calling function. An example decomposition for 4-bit entries is given in figure 2.4(a).



(a)



(b)

Figure 2.4: Decomposition Examples. (a) An example of a decomposition into groups of a matrix M with positive 4-bit entries, with $m_b = 4$, $m_d = 2$ and $k = 4$. (b) An example of a decomposition into groups of a matrix M with negative 4-bit entries, where $m_b = 4$, $m_d = 2$, and $k = 4$.

$10001 = -15$	$01111 = 15$
$101 = -3 \cdot 2^{0\cdot(3-1)} = -3$	$011 = 3 \cdot 2^{0\cdot(3-1)} = 3$
$100 = -4 \cdot 2^{(1\cdot(3-1)} = -16$	$011 = 3 \cdot 2^{(1\cdot(3-1)} = 12$
$101 = -3 \cdot 2^{2\cdot(3-1)} = -48$	$000 = 0 \cdot 2^{2\cdot(3-1)} = 0$
$+ = -57 \neq -15$	$+ = 15$

Figure 2.5: 2's Complement Decomposition. An example of decomposition failing for an integer stored in 2's complement and its positive equivalent using equation 2.15 with $m_b = 5$, $m_d = 3$, and $k = 3$.

In order for the method outlined above to work, negative entries which are represented internally using 2's complement must be handled slightly differently, as illustrated in figure 2.4(b). The reason is that negative numbers stored internally as 2's complement cannot be broken apart the same way as positive numbers, as shown in figure 2.5. Fortunately this is easily remedied by temporarily reversing the sign of the original entry before decomposing it into its constituent groups. Once the bits needed for a group have been extracted from the original entry, they can be stored with the original sign bit setting (i.e. set to negative again) to ensure that the results of the multiply come out correctly.

It should be noted that when choosing values for m_d and v_d they should be set as large as possible to reduce the number of groups that are needed. The reason is that for every group another I/O operation must take place to load either the group matrix or vector onto the EnLight256, which is very costly. Minimizing the number of groups is the surest way to improve the speed of the multiplications. The caveat is that due to the larger sizes of the entries in the group more partitions (as described in section 2.3.2) will be needed and thus more vector shifting will be done on the hardware (as described in section 2.4.2). However testing has shown that it is preferable to lower the number of groups as the shifting, taking place on the hardware, is much less costly than the matrix and vector I/O. Testing has shown that setting the size of the matrix groups to 4 bits and vector groups to 3 bits is optimal in respect to time required to process the multiply. The hardware timing results for differing values of m_b and v_b in figure 2.6.

2.5.2 pre_decomp

The function `pre_decomp` will determine the number of matrix and vector groups required for the decomposition, as described in equations 2.13 and 2.14, using the number `mbits` of bits used for the entries in the matrix and the number `vbits` of bits used for the entries in the vector being multiplied, and the number `dmbits` of bits used to decompose the matrix and the number `dvbits` of bits used to decompose the vector. The resulting number of groups will then be stored into the storage pointed to by `mb` and `vb` for the matrix and

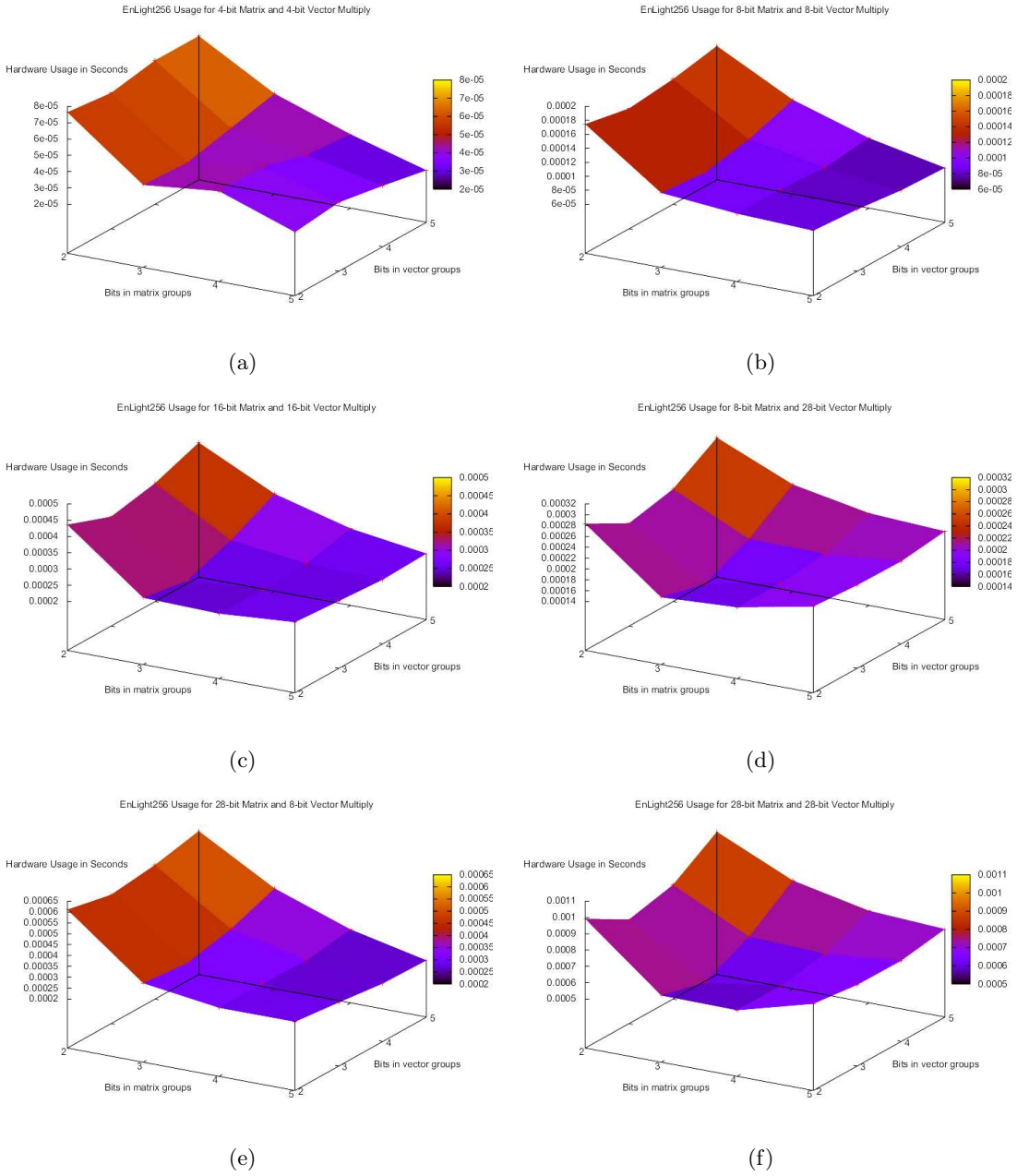


Figure 2.6: Decomposition Timings. Surface plots of decomposition timings for differing bit values in the matrices and vectors, and the decompositions used to multiply them on the EnLight256. Note that data for points with x and y values greater than 7 (i.e. those decompositions that would use more than 7 bits total) was interpolated by the graphing software and is included to make the interpretation of valid data less difficult for the reader.

vector respectively. The method used to determine the number of needed groups is the same as outlined in 2.5.1, but is modified slightly to improve utility as described in 2.6.

2.5.3 decomp_mv

Function `decomp_mv` will decompose the matrix and vector into groups small enough for the `HP_MUL` command to handle using the method described in section 2.5.1 and equation 2.17, where `matrix_type m`, `vec_type v` are the original matrix and vector, `int mdec` and `int vdec` are arrays in which to place the decomposed groups of `m` and `v` into. `int mbits` and `int vbits` are the number of bits in the original `m` and `v`, `int dvbits` and `int dmbits` represent the number of bits to use to create each decomposed group from. `int mb` and `int vb` are the number of matrices and vectors to decompose into as determined by the function `pre-decomp` (described in 2.5.2). This function is later compartmentalized into two separate functions as described in 2.6.

2.6 56bitmm.c

`56bitmm.c` was designed to squeeze the maximum possible number of bits the decomposition method as presented in 2.5.1 is capable of, and can be found in appendix E. Also we used it as an opportunity to further compartmentalize the functions created for earlier programs to allow for greater flexibility in their use. The program will allow a total of 56 bits to be used in the matrix and vector being multiplied, with a maximum of 31 bits used for either. Only 56 bits are allowed due to the fact that the `double int` type, on the architecture on which the software was developed, is 64 bits and is the largest convenient type to store the result matrix/vector multiply. Given this, assuming matrix m with m_b-bit `int` entries and vector v with v_b-bit `int` entries and the possibility of $256 = 2^8$ used entries in any row in m and v, the following inequality arises:

$$2^{m_b} \cdot 2^{v_b} \cdot 2^8 \leq 2^{64} \Rightarrow m_b + v_b \leq 56 \tag{2.18}$$

The reason only 31 bits per entry are allowed in the entries is because the 32^{nd} bit is the sign bit. If a architecture with larger word basic types (i.e. 128-bit `doubles`) was desired, the code is easily modifiable to make use of the extra bits.

2.6.1 pre_decompm

`pre_decompm` is one half of the original `pre_decomp` function (described in section 2.5.2). This half performs the pre-decomposition for a matrix, as described in equation 2.13. `pre_decompm` determines the number of matrix groups that will be required for decomposition using the number of bits used in the original matrix, `int mbits`, and the number of bits that the matrix will be decomposed into,`int dmbits`, and stores the resulting value in `int *mb`.

2.6.2 pre_decompv

`pre_decompv` will perform the other half of the `pre_decomp` function (described in section 2.5.2) by determining the number of vector groups required for decomposition, as described in equation 2.14. Using the number of bits used in the original vector, `int vbits`, and the number of bits to be used to decompose the vector, `int dvbits`, `pre_decompv` determines the required number of groups to ensure the decomposition succeeds and stores the resulting value in `int *vb`.

2.6.3 decomp_m

`decomp_m` performs the decomposition on the matrix into groups as described in equation 2.15. This function was portioned out of the function `decomp_mv` (section 2.5.3) and allows the programmer to decompose a matrix, pointed to by `long **m` with `int ents` many entries, into `int mb` groups of `int dmbits` bits, without forcing the programmer to also provide a vector for decomposition. The decomposed groups are stored in the matrix array pointed to by `matrix_type *mdec`, and the original matrix pointed to by `long **m` remains unchanged.

2.6.4 decomp_v

`decomp_v` will perform the decomposition of the vector into groups as described by equation 2.16. This function was originally part of `decomp_mv` (section 2.5.3) but we compartmentalized it to allow greater flexibility. The function will decompose the vector pointed to by `long *v` (without changing the original vector) into `int vb` many vector groups consisting of `int ents` many entries of `dvbits` bits each. The vector groups are stored in the array pointed to by `vec_type *vdec` for later use.

2.6.5 mshift

`mshift` was initially part of `mvshift` (section 2.4.1), but was pulled out to allow greater flexibility to the programmer by allowing the shifts for the matrix to be done without needing to also apply a shift to a vector. `mshift` takes the number of bits, `int mbits`, used to create the entries of the `matrix_type` matrix `m` and applies the appropriate amount of shift (as determined by equation 2.1) to each entry of `m` and stores the result in the corresponding entry in `matrix_type` matrix `sm`, with the resulting `sm` being a properly shifted matrix that can be passed to the `HP_MUL` command (as described in 2.4.3). In `56bitmm.c`, `mshift` is intended to be used with parts of the original matrix as decomposed by `decomp_m` (described in section 2.6.3).

2.6.6 vshift

Like `mshift`, `vshift` was also initially part of `mvshift` (section 2.4.1), but it too was pulled out to allow greater flexibility to the programmer by allowing the shifts for the vector to be done without needing to also apply a shift to a matrix as well. `vshift` takes the number of bits, `int vbits`, used to create the entries of the `vec_type` vector `v` and applies the appropriate amount of shift (as determined by equation 2.2) to each entry

of v and stores the result in the corresponding entry in vec_type vector sv, with the resulting sv being a properly shifted vector that can be passed to the HP_MUL command (as described in 2.4.3). In 56bitmm.c, vshift is intended to be used with parts of the original vector as decomposed by decomp_v (described in section 2.6.4).

2.6.7 get_vmm_shift

get_vmm_shift was originally the return value of mvshift (section 2.4.1). get_vmm_shift determines the shift to pass the APL_VMM command, s, as determined by equation 2.3. The arguments int mbits and int vbits represent the number of bits used for the entries of the matrix and vector respectively to be used with the APL_VMM command.

2.6.8 MUL_56

Function MUL_56 will, using the decomposition into groups method outlined in 2.5.1, multiply the matrix pointed to by long **m by the vector pointed to by long *v. MUL_56 requires that the total number of bits used for each entry in the matrix and each entry in the vector be ≤ 56, and also that the number of bits used for the entries in both the matrix and vector be less than 32. The reasons for the second limitation are that the long int type uses 32 bits of storage, and that the 32^{nd} bit of any entry is used to store the sign. The vector result of the matrix/vector multiply is stored in the array pointed to by long long *ans.

The arguments int mbits and int vbits allow the user to specify the number of bits used for the entries of the matrix and vector respectively. dmbits and dvbits allow the user to specify the number of bits to decompose the matrix and vector into when the groups are created for the multiply. int num_entries specifies the number of entries in the matrix and vector (which must be ≤ 256).

2.6.9 get_mbits

get_mbits will run through all of the entries in the int r $\times$ c matrix pointed to by long **m and find the largest absolute entry. It will use the value of that entry to determine the maximum number of bits, m_b, used by entries in the matrix and return $m_b + 1$, where the $+1$ is added to ensure compatibility with all other functions in the library. This is useful in the event that the number of bits used to generate the entries in the matrix to be multiplied is not known before hand, and due to the necessity of this information for using many of the other functions in the library.

2.6.10 get_vbits

get_vbits will run through all of the entries in the int r length vector pointed to by long *v and find the largest absolute entry. It will use the value of that entry to determine the maximum number of bits, v_b, used by entries in the vector and return $v_b + 1$, where the $+1$ is added to ensure compatibility with all other functions in the library. This is useful in the event that the number of bits used to generate the entries in the vector is

not known before hand, and due to the necessity of this information for using many of the other functions in the library.

Chapter 3

The Jacobi Method on the EnLight256

In order to test EnLight256 in a realistic situation using high-precision numbers it was decided to implement the Jacobi Method on the Enlight256 using the high-precision tool kit developed for this thesis.

3.1 Description of the Jacobi Method

The Jacobi Method is a simple algorithm that allows, given $n \times n$ matrix A and length n vector b, the vector x to be solved for in the equation $Ax = b$. It iteratively solves for the vector x. The pseudo code for the Jacobi Method can be seen in figure 3.1 (from [Dongarra, 1995]).

3.2 First EnLight256 Implementation

In order to implement the Jacobi Method on the EnLight256 the program `jacobi_enlight.c` was written, the code for which can be found in appendix F. The algorithm is essentially that outlined in figure 3.1, however the two inner `for` loops have been replaced with a call to `MUL_56`.

The high-precision tool kit library for the EnLight256 used in this implementation of the Jacobi Method consists of those functions that are described in section 2.6, and is included in the file `enlight256_hp.h`, which is included in its entirety in appendix G. The functions `gen_m` and `gen_v` simply generate a random diagonally dominant matrix and vector with the number of desired bits. The function `check_convergence` will check the vector x for convergence within a relative tolerance which is supplied through the command line.

3.3 Results of the First EnLight256 Jacobi Implementation

In figure 3.2 the results of various timings for the EnLight256 Jacobi Method implementation are graphed against comparable results for a standard single processor implementation

Choose an initial guess $x^{(0)}$ to the solution x.
for $k = 1, 2, \ldots$
  for $i = 1, 2, \ldots, n$
    $\bar{x}_i = 0$
    for $j = 1, 2, \ldots, i-1, i+1, \ldots, n$
      $\bar{x}_i = \bar{x}_i + a_{ij} x_j^{(k-1)}$
    end
    $\bar{x}_i = (b_i - \bar{x}_i)/a_{ii}$
  end
  $x_i^{(k)} = \bar{x}$
  check convergence; continue if necessary.
end

Figure 3.1: Jacobi Method Pseudo Code. A brief overview of the code used to implement the Jacobi Method.

(as outlined in figure 3.1). Even using an un-optimized high-level function from the tool kit (`MUL_56`), the EnLight256 is much faster than the single processor for 12-bit entries as shown in 3.2(a), and is comparable for 28-bit entries, as shown in 3.2(b). It should also be noted that the Jacobi Method is also implementable in a parallel computing environment due to the way each element of x is calculated separately.

3.4 A Second EnLight256 Implementation

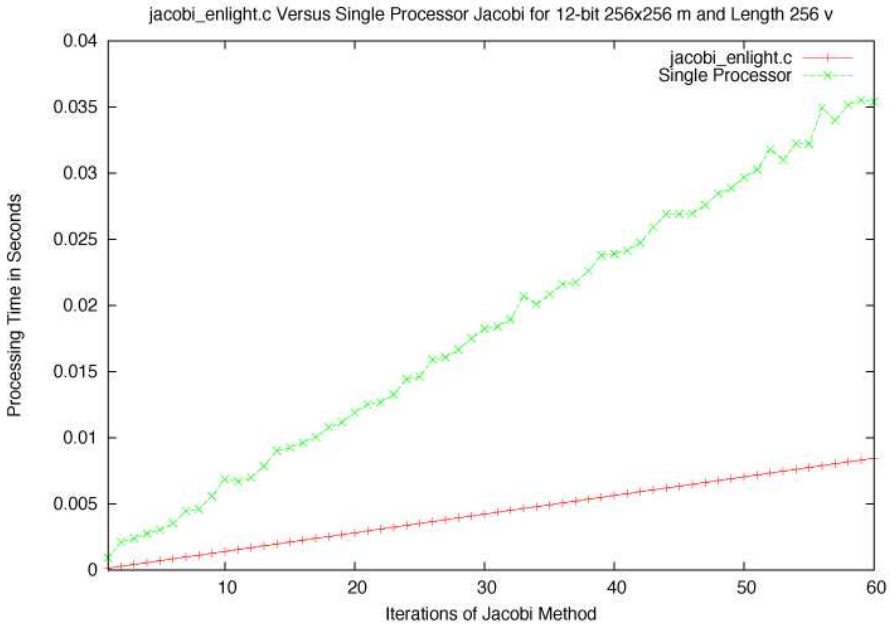
Given that in the previous implementation of the Jacobi Method on the EnLight256 much of the total processing time for the hardware was due to I/O, it was decided to limit the number of bits in the matrix being multiplied to lessen the number of needed matrix loads. This was done in order to allow for a decomposition of the matrix into a maximum of 4 groups, which would allow each group to be loaded into the EnLight256 once using the 4 matrix registers, thus requiring only 4 matrix loads.

For the second EnLight256 implementation using the developed tool kit, the maximum number of bits allowed for the entries of the matrix was determined by setting k from equation 2.13 to 4, and m_d to 4 as well. Solving for m_b gives $m_b = 12$. Thus a maximum of 12 bits are allowed for `jacobi_enlight2.c`, which can be found in appendix H, and at most 4 matrix groups will arise from the decomposition of matrix m and there will be enough matrix registers on the hardware to hold them.

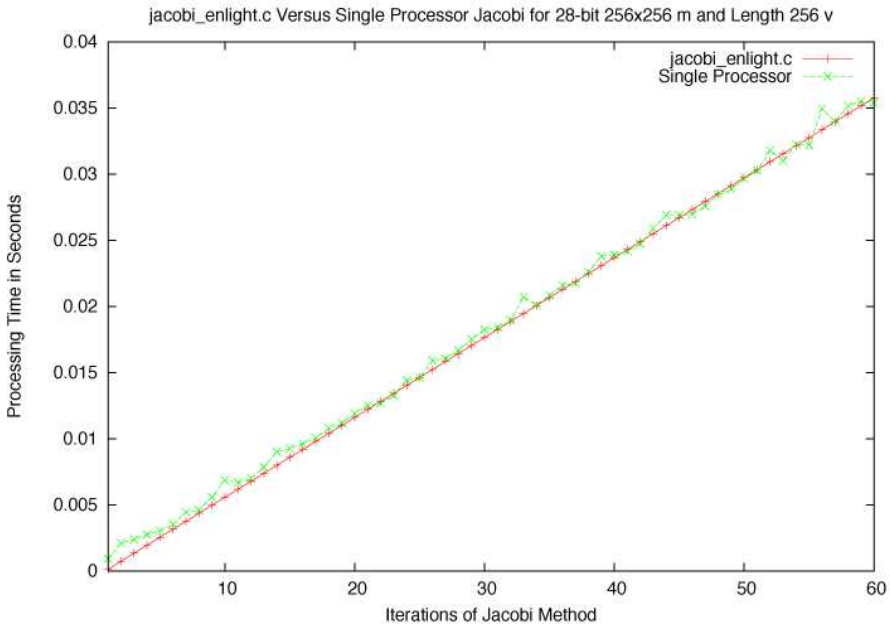
The `jacobi_enlight2.c` implementation uses the `HP_MUL` function to multiply the matrix by the vector on the hardware. The implementation is essentially that of the `MUL_56` function, as described in section 2.6.8, with the main difference that the matrix groups are loaded onto the hardware only one time.

3.5 Results of the Second EnLight256 Jacobi Implementation

The second attempt at implementing the Jacobi Method on the Enlight256 hardware showed an appreciable improvement in processing time at the cost of a lower maximum precision. As shown in figure 3.3 for 12 bit matrices and vectors, `jacobi_enlight2.c` requires significantly less processing time than that which `jacobi_enlight.c` requires. While this is certainly an improvement, it is unfortunate that the precision must be lowered to only 4 bits greater than the EnLight256's native precision. In order to lower the processing time further it would be necessary to reduce the number of vector loads, hardware shitfs, ANDs and actual `APL_VMM` multiplies that are being done.



(a)



(b)

Figure 3.2: Jacobi Method EnLight256 Timings. (a) The timings for a single processor implementation of the Jacobi Method with a 256×256 matrix and 256 entry vector against the jacobi_ enlight.c implementation using 12-bits for 60 iterations. The timings for the single processor implementation were run on a 1.5GHz PowerPC G4 processor, running Mac OS X 10.4.2, and were compiled with gcc 4.0. (b) The timings for a 256×256 28-bit matrix and an 256 entry 28-bit vector over 60 iterations using the EnLight256 versus the single processor implementation.

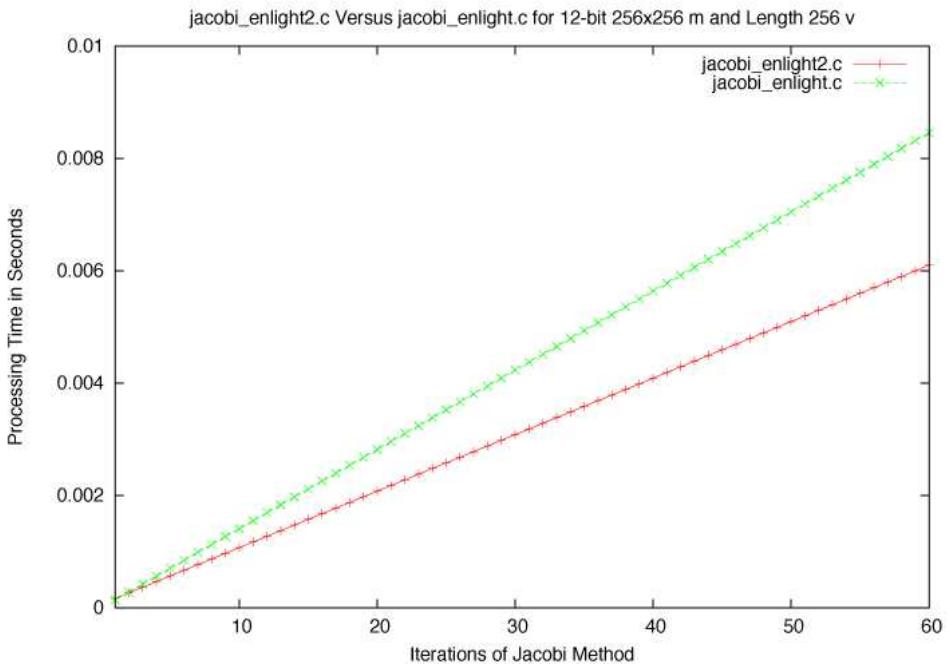


Figure 3.3: Timings for the First and Second Jacobi Implementations. Timings for jacobi_enlight2.c and jacobi_enlight.c for 12-bit matrices and vectors for 1 to 60 iterations.

Bibliography

Bibliography

[Dongarra, 1995] Dongarra, J. (1995). *The Jacobi Method*. University of Tennessee, http://netlib2.cs.utk.edu/linalg/html_templates/node12.html.

[Grimmell, 2005] Grimmell, B. (2005). *Methods for Achieving Increased Precision from Imprecise Matrix/Vector Multipliers*. Oak Ridge National Laboratories, Oak Ridge, TN.

[Vose, 2004] Vose, M. D. (2004). *A C Simulator for the EnLight256*. University of Tennessee, Knoxville, TN.

Appendices

Appendix A

onebit.c:

```
#define MAT_SIZE 7
#define VMM_SHIFT_AMNT 3
#define RAND_LIM 2

#include "random32.h"
#include "simulate.c"

/*this is just to print data in vectors to screen*/
void print_vec(int num_els, char *message, unsigned long long *print_me){
 int i;

 if(message)
  printf("\n %s\n", message);

 for(i = 0; i < num_els; i++){
     if((i>0)&&(!(i%16)))
      printf("\n");
     printf("%3llu ", print_me[i]);
 }
    printf("\n");
}

int main(int argc, char *argv[]){
 matrix_type ones_M, ones_M2, ones_M3;
 signed char ones_v[256], temp_v[256];
 int i, j, k;
 unsigned long long EL256_result[256], real_result[256];
 Ulong seed;

 if(argc != 2){
  fprintf(stderr, "USAGE: onebit rand_seed\n");
  exit(0);
 }
 sscanf(argv[1], "%lu", &seed);

 /*initialize random number generator*/
 initrand(seed);

 /*fill matrix and vector with random bits either 1 or 0, zero out unused portion*/
 for(i = 0; i < 256; i++){
  if(i < MAT_SIZE){
```

```
   ones_v[i] = rnd(RAND_LIM);}
  else
   ones_v[i] = 0;
   real_result[i] = 0;
  for(j = 0; j < 256; j++){
     if((i < MAT_SIZE)&&(j < MAT_SIZE)){
      if(j < MAT_SIZE/2){
       ones_M[i][j] = rnd(RAND_LIM);
       ones_M2[i][j] = 0;
      }
      else{
       ones_M[i][j] = 0;
       ones_M2[i][j] = rnd(RAND_LIM);
      }
     }
     else{
       ones_M[i][j] = 0;
       ones_M2[i][j] = 0;
     }
  }
 }

SMSET(ones_M, 0);
SMSET(ones_M2, 1);
SVSET(ones_v, 1);
print_m(0, 0, 0, MAT_SIZE, MAT_SIZE/2, "matrix half 1", "%d");
print_m(1, 0, MAT_SIZE/2, MAT_SIZE, ((MAT_SIZE/2)?MAT_SIZE/2+1:MAT_SIZE/2),\
             "matrix half 2", "%d");
print_s(1, "VECTOR MULTIPLIER:", "%d");


 /*determine the real answer (yeah i know its disgustingly inefficient)*/
 for(i = 0; i < 256; i++){
  for(j = 0; j < 256; j++){
     if(j < MAT_SIZE/2){
       real_result[i]+= ones_M[i][j]*ones_v[j];
     }
     else{
       real_result[i]+= ones_M2[i][j]*ones_v[j];
     }
  }
 }

 /*perform adjustments on the matrix and vector to overcome 15bit shift
   applied to all APL_VMM (matrix/vector multiplies)
   NOTE: since im just doing this program for single bit entries
         i'm just gonna use a for loop to do it for both the
         matrix and vector*/

 printf("\nM AND v SHIFTING BY %d\n", SHIFT_AMNT);
 for(i=0; i < 256; i++){
   ones_v[i] = ones_v[i] << SHIFT_AMNT;
   for(j=0; j < 256; j++){
     ones_M[i][j] = ones_M[i][j] << SHIFT_AMNT;
     ones_M2[i][j] = ones_M2[i][j] << SHIFT_AMNT;
```

```
  }
 }

 /*read the matrices then vector into the EnLight's registers*/
 SMSET(ones_M, 0);
 SMSET(ones_M2, 1);
 SVSET(ones_v, 13);

 /*make the enlight multiply first matrix and store result*/
 printf("\nVMM SHIFTING BY %d\n", VMM_SHIFT_AMNT);
 APL_VMM(0, 13, 12, VMM_SHIFT_AMNT); /*M, v, Mv, shift-left*/
 SVEC(12, ones_v);

 /*make the enlight multiply second matrix and store result*/
 APL_VMM(1, 13, 12, VMM_SHIFT_AMNT); /*M, v, Mv, shift-left*/
 SVEC(12, temp_v);

 /*sum the two vector results*/
 for(i=0; i<MAT_SIZE; i++)
   EL256_result[i] = temp_v[i] + ones_v[i];

 print_vec(MAT_SIZE, "EL256 ANSWER:", EL256_result);
 print_vec(MAT_SIZE, "REAL ANSWER:", real_result);

 /*compare real answer to enLight answer*/
 for(i = 0; i < MAT_SIZE; i++){
  if(EL256_result[i] != real_result[i]){
   printf("\nREAL ANSWER AND ENLIGHT ANSWER DO NOT MATCH!!\n");
   exit(0);
  }
 }

 printf("\nREAL ANSWER == ENLIGHT ANSWER!! AWWW YEAAHHH!\n");

 return 0;
}/*END OF MAIN*/
```

Appendix B

```
/*1-3bit.c
  This programn will fill a nxn matrix and n length vector randomly with
  entries of up to 3 bits in size, where the maximum number of bits and
  N are declared on the command line.
  NOTES:
   -this program will determine the number of matrix partitions required
    to ensure no one partition is too large for the EnLight256 when
    multiplied. This will ensure no precision is lost in the final
    result due to the hard-limiting on the EnLight256
   -mask will be applied to the vector before multiplication to ensure
    only the entries required for the next partition to be multiplied are
    used. All other entries in the vector will be zeroed by the mask to effect
    this. This is good because it is much more efficient than using many
    matrices to represent the partitions, thus wasting space to store the
    unused space in each partition, and costly in time for filling them
    in.
 */

#include "random32.h"
#include "simulate.c"
#define MAX_SIZE 256

typedef signed char vec_type[MAX_SIZE];

/*this is just to print data in vectors to screen*/
void print_vec(int num_els, char *message, long long *print_me)
{
 int i;

 if(message)
  printf("\n %s\n", message);

 for(i = 0; i < num_els; i++){
  if((i>0)&&(!(i%16)))
   printf("\n");
   printf("%lld ", print_me[i]);
 }

 printf("\n");
}/*END OF print_vec*/

void usage()
{
```

```
 fprintf(stderr, "USAGE: onebit num_bits mat_vec_size rand_seed\n\n");
 fprintf(stderr, " num_bits =     number of bits to be used for the random entries\n");
 fprintf(stderr, "                in the matrix. Must be 1, 2, or 3.\n");
 fprintf(stderr, " mat_vec_size = number of entries in the matrix and the vector\n");
 fprintf(stderr, "                min 1, max 256\n");
 fprintf(stderr, " rand_seed =    positive integer used to generate random entries\n\n");
 exit(0);
}/*END OF usage*/

int main(int argc, char *argv[])
{
 matrix_type the_matrix;
 vec_type *vec_partition_mask, the_vector, partial_result;
 int i, j, k, shift_amt, num_bits, num_entries, count, set_count;
 int map_val = 0, rand_range = 1, num_partitions = 1, temp = 1;
 int min_ent_per_part, max_ent_per_part, leftover_ents, vmm_shift, mv_shift;
 float min_ent_prec;
 long long EL256_result[MAX_SIZE], real_result[MAX_SIZE];
 Ulong seed;

 /*read & error check arguments*/
 if(argc != 4){
  usage();
 }
 sscanf(argv[1], "%d", &num_bits);
 if((num_bits < 1)||(num_bits > 4)){
  usage();
 }
 sscanf(argv[2], "%d", &num_entries);
 if((num_entries < 1)||(num_entries > MAX_SIZE)){
  usage();
 }

 sscanf(argv[3], "%lu", &seed);

 /*initialize random number generator*/
 initrand(seed);

 /*rand_range determines the maximum number returned by the
   random number generator. map_val is subtracted from the
   generated random numbers to map them to the appropriate
   bit space*/
 rand_range = rand_range << num_bits;
 if(!(num_bits == 1))
  map_val = ++map_val << (num_bits - 1);

 /*fill matrix and vector with random numbers, zero out unused portion*/
  for(i = 0; i < MAX_SIZE; i++){
   if(i < num_entries){
    the_vector[i] = rnd(rand_range) - map_val;
   }
   else
    the_vector[i] = 0;

    real_result[i] = 0;
```

```
   EL256_result[i] = 0;
  for(j = 0; j < MAX_SIZE; j++){
   if((i < num_entries)&&(j < num_entries)){
     the_matrix[i][j] = rnd(rand_range) - map_val;
   }
   else{
    the_matrix[i][j] = 0;
   }
  }
 }

/*determine the real answer (yeah i know its disgustingly inefficient)*/
for(i = 0; i < num_entries; i++){
 for(j = 0; j < num_entries; j++){
   real_result[i]+= the_matrix[i][j]*the_vector[j];
 }
}

SMSET(the_matrix, 0);
SVSET(the_vector, 0);

print_m(0, 0, 0, num_entries, num_entries, "matrix", "%d");
print_s(0, "VECTOR MULTIPLIER:", "%d");

/*determine the number of partitions to break matrix into*/
temp = 9 - (2*num_bits);
max_ent_per_part = 1;
max_ent_per_part = (max_ent_per_part << temp) -1;

num_partitions = MAX_SIZE/max_ent_per_part;
min_ent_prec = (float) MAX_SIZE/num_partitions;
if(((float)(min_ent_prec-max_ent_per_part))> 0.0){
   num_partitions++;
   min_ent_prec = (float) MAX_SIZE/num_partitions;
}

min_ent_per_part = MAX_SIZE/num_partitions;
leftover_ents = MAX_SIZE%num_partitions;

printf("Matrix will be split into %d partitions with max %d, min %d entries\n\n",\
       num_partitions, max_ent_per_part, min_ent_per_part);

/*malloc and appropriately fill a vector mask for each partition*/
vec_partition_mask = (vec_type *)malloc(sizeof(vec_type)*num_partitions);
if(!vec_partition_mask){
 fprintf(stderr, "NO MEMORY FOR VECTOR BIT MASKS!! EXITING...\n\n");
 exit(1);
}
for(i = 0; i < num_partitions; i++){
 set_count = 0;
 for(j = 0; j < MAX_SIZE; j++){
  if(j == count){
   if(set_count < min_ent_per_part){
    vec_partition_mask[i][count] = -1;
    count++;
```

```
    set_count++;
   }
   else if((leftover_ents > 0)&&(set_count == min_ent_per_part)){
    vec_partition_mask[i][count] = -1;
    count++;
    set_count++;
    leftover_ents--;
   }
  }
  else
   vec_partition_mask[i][j] = 0;
 }
 printf("vector mask %d gets %d entries\n", i, set_count);
}

/*determine the shifts to apply*/
mv_shift = 8 - num_bits;
vmm_shift = 15 - (2*mv_shift);

printf("\nSHIFTING BY %d ON M, v, %d in APL_VMM\n\n", mv_shift, vmm_shift);

/*apply max shift to all entries in matrix and vector then store both
  for use in multiplication*/
for(i = 0; i < num_entries; i++){
 the_vector[i] = the_vector[i] << mv_shift;
 for(j = 0; j < num_entries; j++){
  the_matrix[i][j] = the_matrix[i][j] << mv_shift;
 }
}
SMSET(the_matrix, 0);
SVSET(the_vector, 0);

/*multiply the matrix by the vector with the approprite mask applied
  and add the resultant vector to the total result*/
for(i = 0; i < num_partitions; i++){
 SVSET(vec_partition_mask[i], 1);
 /*AND the mask with the vector and place the result into VMM vector*/
 APL_AND(0, 1, 13);
 APL_VMM(0, 13, 12, vmm_shift); /*M, v, Mv, shift-left*/

 /*print_s(1, "VECTOR MULTIPLIER:", "%d");*/

 /*store the result*/
 SVEC(12, partial_result);
 for(j = 0; j < num_entries; j++){
  EL256_result[j] += partial_result[j];
 }

}

print_vec(num_entries, "EL256 ANSWER:", EL256_result);
print_vec(num_entries, "REAL ANSWER:", real_result);

/*compare real answer to enlight answer*/
for(i = 0; i < num_entries; i++){
```

```
  if(EL256_result[i] != real_result[i]){
    printf("\nREAL ANSWER AND ENLIGHT ANSWER DO NOT MATCH!!\n");
    exit(0);
  }
 }
 printf("\nREAL ANSWER == ENLIGHT ANSWER!! AWWW YEAAHHH!\n");

 return 0;
}/*END OF MAIN*/
```

Appendix C

```
/*4-3maxmul.c
 This programn will fill a nxn matrix and n length vector randomly with
 entries of up to 4 and 3 bits in size, where the maximum number of bits to
 be used for the matrix and vector and n are declared on the command line.
NOTES:
 1-3bit.c
 -this program will determine the number of matrix partitions required
 to ensure no one partition is too large for the EnLight256 when
 multiplied. This will ensure no precision is lost in the final
 result due to the hard-limiting on the EnLight256
 -mask will be applied to the vector before multiplication to ensure
 only the entries required for the next partition to be multiplied are
 used. All other entries in the vector will be zeroed by the mask to effect
 this. This is good because it is much more efficient than using many
 matrices to represent the partitions, thus wasting space to store the
 unused space in each partition, and costly in time for filling them
 in.
 4-3maxmul.c
 - this program will now do the vector summation on the hardware
   rather than in the calling c function
 - the components have been appropriately functioned off
 - only one vector mask will now be created and then shifted
   on the hardware to multiply the next partition (as opposed
   to creating many vector masks and loading each as done in
   1-3bit.c
 - one more bit of precision is allowed in this version of the
   multiplier. A maximum of 7 bits may be used with the number
   of bits used for the matrix and vector not exceeding 4
*/

#include "random32.h"
#include "simulate.c"

/*declare the max matrix/vector size and the registers you'd like
  to use on the EnLight256. Note that the SUM_REG refers to a
  long vector register (thus takes up two concurrent short vector
  registers), M_REG to a matrix register, and MASK_REG and
  V_REG are short vector registers*/
#define MAX_SIZE 256
#define MASK_REG 6
#define M_REG 0
#define V_REG 0
#define SUM_REG 2
```

```
typedef signed char vec_type[MAX_SIZE];

/*this is just to print data in vectors to screen*/
void print_vec(int num_els, char *message, short *print_me)
{
 int i;

 if(message)
  printf("\n %s\n", message);

 for(i = 0; i < num_els; i++){
  if((i>0)&&(!(i%16)))
   printf("\n");
  printf("%4hd ", print_me[i]);
 }

 printf("\n");
}/*END OF print_vec*/


void usage()
{
 fprintf(stderr, "USAGE: 4-3maxmul num_bits_mat num_bits_vec mat_vec_size rand_seed\n\n");
 fprintf(stderr, " num_bits_mat = number of bits to be used for the random entries\n");
 fprintf(stderr, " in the matrix. Must be 1 through 4 and num_bits_mat+num_bits_vec <= 7\n");
 fprintf(stderr, " num_bits_vec = number of bits to be used for the random entries\n");
 fprintf(stderr, " in the vector. Must be 1 through 4 and num_bits_mat+num_bits_vec <= 7.\n");
 fprintf(stderr, " mat_vec_size = number of entries in the matrix and the vector\n");
 fprintf(stderr, " min 1, max 256\n");
 fprintf(stderr, " rand_seed = positive integer used to generate random entries\n\n");
 exit(0);
}/*END OF usage*/


/*gen_mv will fill in the passed in matrix and vector with appropriate
  values (determined by the values in mbits and vbits)*/
void gen_mv(matrix_type m, vec_type v, int mbits, int vbits, int num_entries)
{
 int map_val_m = 0, map_val_v = 0, range_m, range_v, i, j;

 /*range determines the maximum number returned by the
 random number generator. map_val is subtracted from the
 generated random numbers to map them to the appropriate
 bit space*/
 range_m = 1 << mbits;
 if(!(mbits == 1))
  map_val_m = ++map_val_m << (mbits - 1);

 range_v = 1 << vbits;
 if(!(vbits == 1))
  map_val_v = ++map_val_v << (vbits - 1);

 /*fill matrix and vector with random numbers, zero out unused portion*/
 for(i = 0; i < MAX_SIZE; i++){
```

```
  if(i < num_entries){
   v[i] = rnd(range_v) - map_val_v;
  }
  else
   v[i] = 0;

  for(j = 0; j < MAX_SIZE; j++){
   if((i < num_entries)&&(j < num_entries)){
    m[i][j] = rnd(range_m) - map_val_m;
   }
   else{
    m[i][j] = 0;
   }
  }
 }

}/*END OF gen_mv*/


/*gen_vmask will generate the vector mask applied to the
  vector to be multiplied by the matrix in HP_MUL.*/
void gen_vmask(int mbits, int vbits, vec_type v_mask)
{
 int max_ent, num_partitions, k, i;

 if(v_mask == NULL){
  fprintf(stderr, "v_mask in gen_vmask not initialized. Exiting...\n");
  exit(1);
 }
 if((mbits < 1)||(vbits < 1)){
  fprintf(stderr, "matrix and vector must be composed of at least 1 bit entries!!\n");
  exit(1);
 }

 /*determine the number of partitions to break matrix into*/
 max_ent = (1<<(9-(mbits + vbits))) - 1;
 num_partitions = (MAX_SIZE/max_ent) + 1;

 /*create the vector partition mask*/
 for(k = num_partitions, i = 0; i < MAX_SIZE; i++){

  if(num_partitions == k){
   v_mask[i] = -1;
   k = 1;
  }
  else{
   v_mask[i] = 0;
   k++;
  }
 }

}/*END OF gen_vmask*/


/*mvshift will determine and apply the appropriate shift to the vector v and
```

```
  matrix m
  and store the shifted results in sv and sm respectively,
  and return the amount of shift to be applied to the APL_VMM command*/
int mvshift(int mbits, int vbits, matrix_type m, matrix_type sm, vec_type v, vec_type sv)
{
 int mshift, vshift, vmm_shift, i, j;

 if(m == NULL){
  fprintf(stderr, "m in mvshift not initialized. Exiting...\n");
  exit(1);
 }
 if(sm == NULL){
  fprintf(stderr, "m in mvshift not initialized. Exiting...\n");
  exit(1);
 }
 if(v == NULL){
  fprintf(stderr, "m in mvshift not initialized. Exiting...\n");
  exit(1);
 }
 if(sv == NULL){
  fprintf(stderr, "m in mvshift not initialized. Exiting...\n");
  exit(1);
 }
 if((mbits < 1)||(vbits < 1)){
  fprintf(stderr, "matrix and vector must be composed of at least 1 bit entries!!\n");
  exit(1);
 }
 if((mbits + vbits) > 7){
  fprintf(stderr, "the sum of the number of bits in the matrix and vector can not be \
                   greater than 7 for mvshift!!\n");
  exit(1);
 }

 /*determine the shifts to apply*/
 mshift = 8 - mbits;
 vshift = 8 - vbits;
 /*following two conditionals added to avoid sign bit causing problems in
    multiplication*/
 if(mshift == 7)
  mshift--;
 if(vshift == 7)
  vshift--;
 vmm_shift = 15 - (mshift + vshift);

/*apply max shift to all entries in matrix and vector then store both
  for use in multiplication*/
 for(i = 0; i < MAX_SIZE; i++){
  sv[i] = v[i] << vshift;
  for(j = 0; j < MAX_SIZE; j++){
   sm[i][j] = m[i][j] << mshift;
  }
 }

 return vmm_shift;
```

```
}/*END OF mvshift*/


/*HP_MUL will multiply the passed in matrix by the passed in vector.
  The matrix, vector and vector mask are passed in as EnLight256 register numbers,
  and the mask is used to partition the matrix to ensure no precision is lost.
  Note that while m, v and mask_reg must point to initialized vectors and matrix,
  the sum_reg argument only tells the command what register to use for summing
  the partial results internally and need no be initialized. mbits and
  vbits represent the maximum number of bits in the matrix and vector
  respectively, vmm_shift is the shift passed to the APL_VMM command*/
short int *HP_MUL(int m, int v, int mask_reg, int sum_reg, int mbits, \
                  int vbits, int vmm_shift)
{
 int i, k;
 int max_ent_per_part, num_partitions;
 short int *v_sum;

 if((m < 0)||(v < 0)||(mask_reg < 0)||(sum_reg < 0)||(m > 3)||(v > 15)|| \
                    (mask_reg > 15)||(sum_reg > 7)){
  fprintf(stderr, "Invalid register supplied to HP_MUL. Exiting...\n");
  exit(1);
 }
 if((mbits < 1)||(vbits < 1)){
  fprintf(stderr, "matrix and vector in HP_MUL must be composed of at least \
                   1 bit entries!!\n");
  exit(1);
 }
 if((vmm_shift < 0)||(vmm_shift > 6)){
  fprintf(stderr, "Invalid shift supplied to HP_MUL!! Must be between 0 and \
                   6, got %d!\n", vmm_shift);
  exit(1);
 }
 i = sum_reg * 2;
 k = i++;
 if((v == mask_reg)||(v == i)||(v==k)||(mask_reg == i)||(mask_reg == k)){
  fprintf(stderr, "Invalid register combination passed to HP_MUL!\n");
  fprintf(stderr, "v = %d, mask_reg = %d, sum_reg = %d = %d and %d short registers \
                  used\n.", v, mask_reg, sum_reg, i, k);
  exit(1);
 }
 if((v == 12)||(v == 13)||(v==9)||(mask_reg == 13)||(mask_reg == 12)||(sum_reg==5)\
             ||(sum_reg == 6)){
  fprintf(stderr, "Short Vector Registers 12 and 13 are used internally in HP_MUL\n");
  fprintf(stderr, "Make sure v, mask_reg and sum_reg do not use these registers.\n ");
  fprintf(stderr, "Also, only mask_reg can be register 9 as it is used internally. \
                   Exiting...\n");
  exit(1);
 }

 v_sum = (short int*)malloc(sizeof(short int)*MAX_SIZE);

 if(mask_reg != 9)
  APL_COPY(mask_reg, 9);
```

```
 /*determine the number of partitions to break matrix into*/
 max_ent_per_part = (1<<(9-(mbits + vbits))) - 1;
 num_partitions = (MAX_SIZE/max_ent_per_part) + 1;

 /*zero out sum vector for safety*/
 for(i = 0; i < MAX_SIZE; i++){
  v_sum[i] = 0;
 }

 /*initialize the internal result vector to all zero*/
 DVSET(v_sum, sum_reg);

 /*multiply the matrix by the vector with the approprite mask applied
 and add the resultant vector to the total result*/
 for(i = 0; i < num_partitions; i++){

  /*AND the mask with the vector and place the result into VMM vector*/
  APL_AND(v, 9, 13);
  APL_VMM(m, 13, 12, vmm_shift); /*M, v, Mv, shift-left*/

  /*sum the partial result internally*/
  APL_VADDM(sum_reg, 12, sum_reg);

  /*shift the vector mask to multiply the next partition*/
  APL_SHFT_U(9, 9);

 }
 /*store the internally summed answer*/
 DVEC(sum_reg, v_sum);

 return v_sum;

}/*END OF HP_MUL*/


int main(int argc, char *argv[])
{
 matrix_type m, shifted_m;
 vec_type v, v_mask, shifted_v;
 int i, j, k, mbits, vbits, num_entries, vmm_shift;
 short int *v_sum;
 Ulong seed;

 /*read & error check arguments*/
 if(argc != 5){
  usage();
 }
 sscanf(argv[1], "%d", &mbits);
 if((mbits < 1)||(mbits > 4)){
  usage();
 }
 sscanf(argv[2], "%d", &vbits);
 if(((vbits < 1)||(vbits > 4))||(mbits + vbits > 7)){
  usage();
 }
```

```
 sscanf(argv[3], "%d", &num_entries);
 if((num_entries < 1)||(num_entries > MAX_SIZE)){
  usage();
 }
 sscanf(argv[4], "%lu", &seed);

 /*initialize random number generator*/
 initrand(seed);

 /*create vector mask outside of stress loop since it can be reused*/
 gen_vmask(mbits, vbits, v_mask);

 /*generate a nice matrix and vector*/
 gen_mv(m, v, mbits, vbits, num_entries);

 /*(re)set the vector partition mask*/
 SVSET(v_mask, MASK_REG);

 /*shift the matrix and vector then store them*/
 vmm_shift = mvshift(mbits, vbits, m, shifted_m, v, shifted_v);
 SMSET(shifted_m, M_REG);
 SVSET(shifted_v, V_REG);

 /*perform the multiply*/
 v_sum = HP_MUL(M_REG, V_REG, MASK_REG, SUM_REG, mbits, vbits, vmm_shift);

 /*compare real answer to enlight answer*/
 for(i = 0; i < num_entries; i++){
  k = 0;
  for(j = 0; j < num_entries; j++){
   k += m[i][j]*v[j];
  }
  if(v_sum[i] != k){
    printf("\nREAL ANSWER AND ENLIGHT ANSWER DO NOT MATCH!!\n");
    exit(0);
   }
 }

 printf("\nREAL ANSWER == ENLIGHT ANSWER!! AWWW YEAAHHH!\n");

 return 0;
}/*END OF main*/

/*
gcc -I .-Wall -gdwarf-2 -g3 -o 4-3maxmul 4-3maxmul.c -lm
gcc -I .-Wall -O3 -mfpmath=387 -o 4-3maxmul 4-3maxmul.c -lm
*/
```

Appendix D

```
/*8bitmm.c
 This programn will fill a nxn matrix and n length vector randomly with
 entries of up to 8 bits in size, where the maximum number of bits to
 be used for the matrix and vector and n are declared on the command line
 along with the number of bits (and hence partitions) to use for the
 matrix and vector decompositions.
NOTES:
 - the method used to decompose the initial matrix and
 vector into small enough chunks to be handled by the
 HP_MUL routine is to break each entry up into
 groups of bits and store each  group
 in a correspomding decomposed matrix or vector,
 the sum of which, with the appropriate shifts applied
 will yield the initial matrix and vector
 - negative numbers will be temporarily made positive to
 facilitate their decomposition, as the math does not
 work with 2-complement
 - this program uses the library developed for low bit
 matrix vector multiplies on the EnLight256 hardware
 as defined in 4-3maxmul.c

 - testing has revealed that a high partition low I/O strategy
   is best as far as hardware time used. gnuplot used to plot
   resulting times for each combination of matrix and vector
   bitwise sizes and each possible decomposition

 */

#include "random32.h"
#include "simulate.c"

/*declare the max matrix/vector size and the registers you'd like
  to use on the EnLight256. Note that the SUM_REG refers to a
  long vector register (thus takes up two concurrent short vector
  registers), M_REG to a matrix register, and MASK_REG and
  V_REG are short vector registers*/
#define MAX_SIZE 256
#define MASK_REG 6
#define M_REG 0
#define V_REG 0
#define SUM_REG 2

typedef signed char vec_type[MAX_SIZE];
```

```
/*this is just to print data in vectors to screen*/
void print_vec(int num_els, char *message,  long int *print_me)
{
 int i;

 if(message)
  printf("\n %s\n", message);

 for(i = 0; i < num_els; i++){
  if((i>0)&&(!(i%16)))
   printf("\n");
  printf("%8ld ", print_me[i]);
 }

 printf("\n");
}/*END OF print_vec*/


void usage()
{
 fprintf(stderr, "USAGE: 8bitmm num_bits_mat num_bits_vec mat_vec_size rand_seed\n\n");
 fprintf(stderr, " num_bits_mat = number of bits to be used for the random entries\n");
 fprintf(stderr, " in the matrix. Must be 1 through 8\n");
 fprintf(stderr, " num_bits_vec = number of bits to be used for the random entries\n");
 fprintf(stderr, " in the vector. Must be 1 through 8\n");
 fprintf(stderr, " mat_vec_size = number of entries in the matrix and the vector\n");
 fprintf(stderr, " min 1, max 256\n");
 fprintf(stderr, " rand_seed = positive integer used to generate random entries\n\n");
 exit(0);
}/*END OF usage*/


/*gen_mv will fill in the passed in matrix and vector with appropriate
  values (determined by the values in mbits and vbits)*/
void gen_mv(matrix_type m, vec_type v, int mbits, int vbits, int num_entries)
{
 int map_val_m = 0, map_val_v = 0, range_m, range_v, i, j;

 /*range determines the maximum number returned by the
 random number generator. map_val is subtracted from the
 generated random numbers to map them to the appropriate
 bit space*/
 range_m = 1 << mbits;
 if(!(mbits == 1)){
  map_val_m++;
  map_val_m = map_val_m << (mbits - 1);
 }
 range_v = 1 << vbits;
 if(!(vbits == 1)){
  map_val_v++;
  map_val_v = map_val_v << (vbits - 1);
 }

 /*fill matrix and vector with random numbers, zero out unused portion*/
```

```
 for(i = 0; i < MAX_SIZE; i++){
  if(i < num_entries){
   v[i] = rnd(range_v) - map_val_v;
  }
  else
   v[i] = 0;

  for(j = 0; j < MAX_SIZE; j++){
   if((i < num_entries)&&(j < num_entries)){
    m[i][j] = rnd(range_m) - map_val_m;
   }
   else{
    m[i][j] = 0;
   }
  }
 }

}/*END OF gen_mv*/


/*gen_vmask will generate the vector mask applied to the
  vector to be multiplied by the matrix in HP_MUL.*/
void gen_vmask(int mbits, int vbits, vec_type v_mask)
{
 int max_ent, num_partitions, k, i;

 if(v_mask == NULL){
  fprintf(stderr, "v_mask in gen_vmask not initialized. Exiting...\n");
  exit(1);
 }
 if((mbits < 1)||(vbits < 1)){
  fprintf(stderr, "matrix and vector must be composed of at least 1 bit entries!!\n");
  exit(1);
 }

 /*determine the number of partitions to break matrix into*/
 max_ent = (1<<(9-(mbits + vbits))) - 1;
 num_partitions = (MAX_SIZE/max_ent) + 1;

 /*create the vector partition mask*/
 for(k = num_partitions, i = 0; i < MAX_SIZE; i++){

  if(num_partitions == k){
   v_mask[i] = -1;
   k = 1;
  }
  else{
   v_mask[i] = 0;
   k++;
  }
 }

}/*END OF gen_vmask*/
```

```
/*mvshift will determine and apply the appropriate shift to the vector v and matrix m
  and store the shifted results in sv and sm respectively,
  and return the amount of shift to be applied to the APL_VMM command*/
int mvshift(int mbits, int vbits, matrix_type m, matrix_type sm, vec_type v, vec_type sv)
{
 int mshift, vshift, vmm_shift, i, j;

 if(m == NULL){
  fprintf(stderr, "m in mvshift not initialized. Exiting...\n");
  exit(1);
 }
 if(sm == NULL){
  fprintf(stderr, "m in mvshift not initialized. Exiting...\n");
  exit(1);
 }
 if(v == NULL){
  fprintf(stderr, "m in mvshift not initialized. Exiting...\n");
  exit(1);
 }
 if(sv == NULL){
  fprintf(stderr, "m in mvshift not initialized. Exiting...\n");
  exit(1);
 }
 if((mbits < 1)||(vbits < 1)){
  fprintf(stderr, "matrix and vector must be composed of at least 1 bit entries!!\n");
  exit(1);
 }
 if((mbits + vbits) > 7){
  fprintf(stderr, "the sum of the number of bits in the matrix and vector can not be \
                   greater than 7 for mvshift!!\n");
  exit(1);
 }

 /*determine the shifts to apply*/
 mshift = 8 - mbits;
 vshift = 8 - vbits;
 /*following two conditionals added to avoid sign bit causing problems in multiplication*/
 if(mshift == 7)
  mshift--;
 if(vshift == 7)
  vshift--;
 vmm_shift = 15 - (mshift + vshift);

/*apply max shift to all entries in matrix and vector then store both
  for use in multiplication*/
 for(i = 0; i < MAX_SIZE; i++){
  sv[i] = v[i] << vshift;
  for(j = 0; j < MAX_SIZE; j++){
   sm[i][j] = m[i][j] << mshift;
  }
 }

 return vmm_shift;

}/*END OF mvshift*/
```

```
/*HP_MUL will multiply the passed in matrix by the passed in vector.
  The matrix, vector and vector mask are passed in as EnLight256 register numbers,
  and the mask is used to partition the matrix to ensure no precision is lost.
  Note that while m, v and mask_reg must point to initialized vectors and matrix,
  the sum_reg argument only tells the command what register to use for summing
  the partial results internally and need no be initialized. mbits and
  vbits represent the maximum number of bits in the matrix and vector
  respectively, vmm_shift is the shift passed to the APL_VMM command*/
short int *HP_MUL(int m, int v, int mask_reg, int sum_reg, int mbits, int vbits,\
                  int vmm_shift)
{
 int i, k;
 int max_ent_per_part, num_partitions;
 short int *v_sum;

 if((m < 0)||(v < 0)||(mask_reg < 0)||(sum_reg < 0)||(m > 3)||(v > 15)||(mask_reg > 15)\
     ||(sum_reg > 7)){
  fprintf(stderr, "Invalid register supplied to HP_MUL. Exiting...\n");
  exit(1);

 }
 if((mbits < 1)||(vbits < 1)){
  fprintf(stderr, "matrix and vector in HP_MUL must be composed of at least 1 bit\
                   entries!!\n");
  exit(1);
 }
 if((vmm_shift < 0)||(vmm_shift > 6)){
  fprintf(stderr, "Invalid shift supplied to HP_MUL!! Must be between 0 and 6, got\
                    %d!\n", vmm_shift);
  exit(1);
 }
 i = sum_reg * 2;
 k = i++;
 if((v == mask_reg)||(v == i)||(v==k)||(mask_reg == i)||(mask_reg == k)){
  fprintf(stderr, "Invalid register combination passed to HP_MUL!\n");
  fprintf(stderr, "v = %d, mask_reg = %d, sum_reg = %d = %d and %d short registers used\n.",\
                   v, mask_reg, sum_reg, i, k);
  exit(1);
 }
 if((v == 12)||(v == 13)||(v==9)||(mask_reg == 13)||(mask_reg == 12)||(sum_reg==5)\
     ||(sum_reg == 6)){
  fprintf(stderr, "Short Vector Registers 12 and 13 are used internally in HP_MUL\n");
  fprintf(stderr, "Make sure v, mask_reg and sum_reg do not use these registers.\n ");
  fprintf(stderr, "Also, only mask_reg can be register 9 as it is used internally.\
                   Exiting...\n");
  exit(1);
 }

 v_sum = (short int*)malloc(sizeof(short int)*MAX_SIZE);

 if(mask_reg != 9)
  APL_COPY(mask_reg, 9);
```

```
 /*determine the number of partitions to break matrix into*/
 max_ent_per_part = (1<<(9-(mbits + vbits))) - 1;
 num_partitions = (MAX_SIZE/max_ent_per_part) + 1;

 /*zero out sum vector for safety*/
 for(i = 0; i < MAX_SIZE; i++){
  v_sum[i] = 0;
 }
 /*initialize the internal result vector to all zero*/
 DVSET(v_sum, sum_reg);

 /*multiply the matrix by the vector with the approprite mask applied
 and add the resultant vector to the total result*/
 for(i = 0; i < num_partitions; i++){

  /*AND the mask with the vector and place the result into VMM vector*/
  APL_AND(v, 9, 13);
  APL_VMM(m, 13, 12, vmm_shift); /*M, v, Mv, shift-left*/

  /*sum the partial result internally*/
  APL_VADDM(sum_reg, 12, sum_reg);

  /*shift the vector mask to multiply the next partition*/
  APL_SHFT_U(9, 9);

 }
 /*store the internally summed answer*/
 DVEC(sum_reg, v_sum);

 return v_sum;

}/*END OF HP_MUL*/


/*pre_decomp will determine number of matrices and vectors
  required for the decomposition and store the number
  required in mb and vb for the matrix and vector respectively.
  mbits and vbits are the number of bits used in the original
  m and v, dmbits and dvbits are the desired number of bits
  for the decomposed matrices and vectors*/
void pre_decomp(int mbits, int vbits, int dmbits, int dvbits, int *mb,\
                int *vb)
{
 if((dmbits < 2)||(dmbits > 5)||(dvbits < 2)||(dvbits > 5)||((dmbits +\
     dvbits) > 7)){
  printf("The number of bits used to decompose the matrix and vector must\
         be at\n");
  printf("least 2 and no greater than 5, and the sum of the bits used for\
         the\n");
  printf("matrix and vector must be less than 8 in function pre_decomp.\
         Exiting...\n");
  exit(1);
 }
 if(mb == NULL){
  printf("Argument mb in pre_decomp is invalid. Exiting...\n");
```

```
  exit(1);
 }
 if(vb == NULL){
  printf("Argument vb in pre_decomp is invalid. Exiting...\n");
  exit(1);
 }


 *mb = mbits/(--dmbits);
 if(mbits%dmbits)
  (*mb)++;

 *vb = vbits/(--dvbits);
 if(vbits%dvbits)
  (*vb)++;
}/*ENF OF pre_decomp*/


/*decomp_mv will decompose the matrix and vector into groups small enough
  for the HP_MUL command to handle, m and v are the original matrix and
  vector, mdec and vdec are initialized arrays to place the
  decompsed groups of m and v into. mbits and vbits are the
  number of bits in the originals, dvbits and dmbits the
  number of bits to create each group from. mb and vb are the
  number of matrices and vectors to decompose into*/
void decomp_mv(matrix_type m, vec_type v, matrix_type *mdec, vec_type *vdec, \
       int dmbits, int dvbits, int mb, int vb, int ents)
{
 int i, j, k, t, tn = 0, tt, mmask, vmask;

 if((dmbits < 2)||(dmbits > 5)||(dvbits < 2)||(dvbits > 5)||((dmbits +\
     dvbits) > 7)){
  printf("The number of bits used to decompose the matrix and vector\
          must be at\n");
  printf("least 2 and no greater than 5, and the sum of the bits used\
          for the\n");
  printf("matrix and vector must be less than 8 in function pre_decomp.\
          Exiting...\n");
  exit(1);
 }
 if(m == NULL){
  fprintf(stderr, "m in decomp_mv not initialized. Exiting...\n");
  exit(1);
 }
 if(v == NULL){
  fprintf(stderr, "v in decomp_mv not initialized. Exiting...\n");
  exit(1);
 }
 if(mdec == NULL){
  fprintf(stderr, "mdec in decomp_mv not initialized. Exiting...\n");
  exit(1);
 }
 if(vdec == NULL){
  fprintf(stderr, "vdec in decomp_mv not initialized. Exiting...\n");
  exit(1);
```

```
 }
 if((mb < 1)||(vb < 1)){
  fprintf(stderr, "Decompositions must be of at least 1 group in\
                   decomp_mv!\n");
  fprintf(stderr, " matrix groups = %d, vector groups = %d Exiting...\n",\
                   mb, vb);
  exit(1);
 }
 if((ents < 1)||(ents > 256)){
  fprintf(stderr, "Number of entries must be > 0 and < 256 in decomp_mv!\
                   Exiting...\n");
  exit(1);
 }


 /*create a mask to extract bits*/
 for(vmask = 1, k = 1; k < (dvbits-1); k++, vmask++)
  vmask = vmask << 1;

 for(mmask = 1, k = 1; k < (dmbits-1); k++, mmask++)
  mmask = mmask << 1;

 /*for all v[i] and m[i][j] break up each entry into vb and mb groups
   of dvbits and dmbits respectively */
 for(i = 0; i < ents; i++){
  t = v[i];
  tn = 0;
  /*if entry is negative make it positive while you break it up
to ensure 2's complement doesnt mess with the final sums */
  if(t < 0){
   t *= -1;
   tn = 1;
  }

  for(k = 0; k < vb; k++){
   tt = t&vmask;
   if(tn)
tt *= -1;

   vdec[k][i] = tt;
   t = t >> (dvbits - 1);
  }

  for(j = 0; j < ents; j++){
   t = m[i][j];
   tn = 0;
   if(t < 0){
t *= -1;
tn = 1;
   }

   for(k = 0; k < mb; k++){
    tt = t&mmask;
if(tn)
 tt *= -1;
```

```
    mdec[k][i][j] = tt;
    t = t >> (dmbits - 1);
   }

  }
 }


}/*END OF decomp_mv*/


int main(int argc, char *argv[])
{
 matrix_type m, shifted_m, mdec[8];
 vec_type v, v_mask, shifted_v, vdec[8];
 int i, j, k, h, mbits, vbits, dmbits, dvbits, num_entries, vmm_shift;
 int mb, vb;
 short int *v_sum;
 long int full_sum[MAX_SIZE];
 Ulong seed;

 /*read & error check arguments*/
 if(argc != 5){
  usage();
 }
 sscanf(argv[1], "%d", &mbits);
 if((mbits < 1)||(mbits > 8)){
  usage();
 }
 sscanf(argv[2], "%d", &vbits);
 if(((vbits < 1)||(vbits > 8))){
  usage();
 }
 sscanf(argv[3], "%d", &num_entries);
 if((num_entries < 1)||(num_entries > MAX_SIZE)){
  usage();
 }
 sscanf(argv[4], "%lu", &seed);

 /*initialize random number generator*/
 initrand(seed);

 /*just to get it working...*/
 dmbits = 2; dvbits = 2;

 for(i=0; i < MAX_SIZE; i++)
full_sum[i] = 0;

 /*create vector mask outside of main loop since it can be reused*/
 gen_vmask(dmbits, dvbits, v_mask);

 /*generate a nice matrix and vector*/
 gen_mv(m, v, mbits, vbits, num_entries);
```

```
 /*determine number of matrices and vectors required for the decomposition*/
 pre_decomp(mbits, vbits, dmbits, dvbits, &mb, &vb);

 /*decompose the matrix and vector into groups small enough for
   the HP_MUL command to handle*/
 decomp_mv(m, v, mdec, vdec, dmbits, dvbits, mb, vb, num_entries);

 /*multiply the parts of the matrix by the parts of the vector*/
 for(i = 0; i < mb; i++){
  for(j = 0; j < vb; j++){
   /*(re)set the vector partition mask*/
   SVSET(v_mask, MASK_REG);

   /*shift the matrix and vector then store them*/
   vmm_shift = mvshift(dmbits, dvbits, mdec[i], shifted_m, vdec[j], shifted_v);
   SMSET(shifted_m, M_REG);
   SVSET(shifted_v, V_REG);

   /*perform the multiply*/
   v_sum = HP_MUL(M_REG, V_REG, MASK_REG, SUM_REG, dmbits, dvbits, vmm_shift);

   /*apply the proper shift and add the partial sum to the total vector sum*/
   for(k = 0; k < num_entries; k++){
full_sum[k] += (long int) v_sum[k] << ((i*(dmbits-1))+(j*(dvbits-1)));
   }
  }
 }

 /*compare real answer to enlight answer*/
 for(i = 0; i < num_entries; i++){
  k = 0;
  for(j = 0; j < num_entries; j++){
   k += m[i][j]*v[j];
  }
  if(full_sum[i] != k){
    printf("\nREAL ANSWER AND ENLIGHT ANSWER DO NOT MATCH!!\n");
    exit(0);
   }
 }

 printf("\nREAL ANSWER == ENLIGHT ANSWER!! AWWW YEAAHHH!\n");

 return 0;
}/*END OF main*/


/*
gcc -I .-Wall -gdwarf-2 -g3 -o 8bitmm 8bitmm.c -lm
gcc -I .-Wall -O3 -mfpmath=387 -o 8bitmm 8bitmm.c -lm
*/
```

Appendix E

```
/*56bitmm.c
 This programn will fill a nxn matrix and n length vector randomly with
 entries of up to 31 bits in size, 56 bits total for the matrix and
 vector, where the maximum number of bits to
 be used for the matrix and vector and n are declared on the command line
 along with the number of bits (and hence partitions) to use for the
 matrix and vector decompositions.
NOTES:
 - the method used to decompose the initial matrix and
   vector into small enough chunks to be handled by the
   HP_MUL routine is to break each entry up into
   groups of bits and store each  group
   in a correspomding decomposed matrix or vector,
   the sum of which, with the appropriate shifts applied
   will yield the initial matrix and vector
 - negative numbers will be temporarily made positive to
   facilitate their decomposition, as the math does not
   work with 2-complement
 - this program uses the library developed for low bit
   matrix vector multiplies on the EnLight256 hardware
   as defined in 4-3maxmul.c
 - added get_mbits and get_vbits which are functions that
   will determine and return the number of bits + 1 used in
   the largest absolute entry in the matrix or vector
 - the following functions were split: mvshift into vshift,
   mshift and get_vmm_shift; decomp_mv into decomp_v and decomp_m;
 - test_m and test_v added to check that decompostion of m and
   v are correct
 - testing has revealed that a high partition low I/O strategy
   is best as far as hardware time used
 */

#include "random32.h"
#include "simulate.c"

/*declare the max matrix/vector size and the registers you'd like
  to use on the EnLight256. Note that the SUM_REG refers to a
  long vector register (thus takes up two concurrent short vector
  registers), M_REG to a matrix register, and MASK_REG and
  V_REG are short vector registers*/
#define MAX_SIZE 256
#define MASK_REG 6
#define M_REG 0
```

```
#define V_REG 0
#define SUM_REG 2

typedef signed char vec_type[MAX_SIZE];

/*This call checks to see if the passed in pointer is null and
    prints out an error message and exits if it is.*/
void malloc_error(void *check_null, char *mesg)
{
 if(check_null == NULL){
  fprintf(stderr, "Memory Allocation error in %s\n Exiting...", mesg);
  exit(1);
 }
}/*END OF malloc_error()*/


/*this is just to print data in vectors to screen*/
void print_vec(int num_els, char *message,  long *print_me)
{
 int i;

 if(message)
  printf("\n %s\n", message);

 for(i = 0; i < num_els; i++){
  if((i>0)&&(!(i%16)))
   printf("\n");
  printf("%8ld ", print_me[i]);
 }

 printf("\n");
}/*END OF print_vec*/


/*test_m tests to make sure that the decomposition m of was done
  correctly and no data loss has occurred mb = number of groups,
  m is the original matrix, dm is the decomposed matrices*/
void test_m(long **m, int dmbits, int mb, matrix_type *mdec,\
            int num_entries)
{
 int i, j, k, h;

 for(i = 0; i < num_entries; i++){

  for(j = 0; j < num_entries; j++){
   for(h=0, k=0; k < mb; k++){
    h+= mdec[k][i][j] << (k*(dmbits-1));
   }
   if(h != m[i][j]){
    printf("m[%d] %ld!= %d\n", i, m[i][j], h);
    exit(1);
   }
  }
 }
 printf("M OK!!\n");
```

```
}/*END OF test_m*/


/*test_v tests to make sure that the decomposition of v was done
  correctly and no data loss has occurred. vb = number of groups,
  v is original vector, dv is the decomposed vectors */
void test_v(long *v, int dvbits, int vb, vec_type *vdec,\
            int num_entries)
{
 int i, j, h;

 for(i = 0; i < num_entries; i++){
  for(h=0, j=0; j < vb; j++){
   h+= vdec[j][i] << (j*(dvbits-1));
  }
  if(h != v[i]){
   printf("v[%d] %ld!= %d\n", i, v[i], h);
   exit(1);
  }
 }

 printf("V OK!!\n");

}/*END OF test_v*/


void usage()
{
 fprintf(stderr, "USAGE: 56bitmm num_bits_mat num_bits_vec mat_vec_size rand_seed\n\n");
 fprintf(stderr, " num_bits_mat = number of bits to be used for the random entries\n");
 fprintf(stderr, " in the matrix. num_bits_mat + num_bits_vec must be <= 56 \n");
 fprintf(stderr, " num_bits_vec = number of bits to be used for the random entries\n");
 fprintf(stderr, " in the vector. num_bits_mat + num_bits_vec must be <= 56.\n");
 fprintf(stderr, " mat_vec_size = number of entries in the matrix and the vector\n");
 fprintf(stderr, " min 1, max 256\n");
 fprintf(stderr, " rand_seed = positive integer used to generate random entries\n\n");
 exit(0);
}/*END OF usage*/


/*gen_mv will fill in the passed in matrix and vector with appropriate
  values (determined by the values in mbits and vbits)*/
void gen_mv(long **m, long *v, int mbits, int vbits, int num_entries)
{
 int map_val_m = 0, map_val_v = 0, range_m, range_v, i, j;

 /*range determines the maximum number returned by the
 random number generator. map_val is subtracted from the
 generated random numbers to map them to the appropriate
 bit space*/
 range_m = 1 << mbits;
 if(!(mbits == 1)){
  map_val_m++;
  map_val_m = map_val_m << (mbits - 1);
```

```
 }
 range_v = 1 << vbits;
 if(!(vbits == 1)){
  map_val_v++;
  map_val_v = map_val_v << (vbits - 1);
 }

 /*fill matrix and vector with random numbers, zero out unused portion*/
 for(i = 0; i < MAX_SIZE; i++){
  if(i < num_entries){
   v[i] = rnd(range_v) - map_val_v;
  }
  else
   v[i] = 0;

  for(j = 0; j < MAX_SIZE; j++){
   if((i < num_entries)&&(j < num_entries)){
    m[i][j] = rnd(range_m) - map_val_m;
   }
   else{
    m[i][j] = 0;
   }
  }
 }

}/*END OF gen_mv*/


/*gen_vmask will generate the vector mask applied to the
  vector to be multiplied by the matrix in HP_MUL.*/
void gen_vmask(int mbits, int vbits, vec_type v_mask)
{
 int max_ent, num_partitions, k, i;

 if(v_mask == NULL){
  fprintf(stderr, "v_mask in gen_vmask not initialized. Exiting...\n");
  exit(1);
 }
 if((mbits < 1)||(vbits < 1)){
  fprintf(stderr, "matrix and vector must be composed of at least 1 bit entries!!\n");
  exit(1);
 }

 /*determine the number of partitions to break matrix into*/
 max_ent = (1<<(9-(mbits + vbits))) - 1;
 num_partitions = (MAX_SIZE/max_ent) + 1;

 /*create the vector partition mask*/
 for(k = num_partitions, i = 0; i < MAX_SIZE; i++){

  if(num_partitions == k){
   v_mask[i] = -1;
   k = 1;
  }
  else{
```

```
   v_mask[i] = 0;
   k++;
  }
 }

}/*END OF gen_vmask*/



/*mshift will determine and apply the appropriate shift to the matrix m
  and store the shifted results in sm*/
void mshift(int mbits, matrix_type m, matrix_type sm )
{
 int mshift, i, j;

 if(m == NULL){
  fprintf(stderr, "m in mshift not initialized. Exiting...\n");
  exit(1);
 }
 if(sm == NULL){
  fprintf(stderr, "sm in mshift not initialized. Exiting...\n");
  exit(1);
 }
 if(mbits < 1){
  fprintf(stderr, "matrix must be composed of at least 1 bit entries in mshift!!\n");
  exit(1);
 }

 /*determine the shifts to apply*/
 mshift = 8 - mbits;

 /*following two conditionals added to avoid sign
   bit causing problems in multiplication */
 if(mshift == 7)
  mshift--;

/*apply max shift to all entries in matrix then store
  for use in multiplication */
 for(i = 0; i < MAX_SIZE; i++){
  for(j = 0; j < MAX_SIZE; j++){
   sm[i][j] = m[i][j] << mshift;
  }
 }

}/*END OF mshift*/


/*vshift will determine and apply the appropriate shift to the vector v
  and store the shifted results in sv */
void vshift(int vbits, vec_type v, vec_type sv)
{
 int vshift, i;

 if(v == NULL){
  fprintf(stderr, "v in vshift not initialized. Exiting...\n");
```

```
 exit(1);
 }
 if(sv == NULL){
  fprintf(stderr, "sv in vshift not initialized. Exiting...\n");
  exit(1);
 }
 if(vbits < 1){
  fprintf(stderr, "vector must be composed of at least 1 bit entries in vshift!!\n");
  exit(1);
 }

 /*determine the shifts to apply */
 vshift = 8 - vbits;

 /*following two conditional added to
   avoid sign bit causing problems in multiplication */
 if(vshift == 7)
  vshift--;

 /*apply max shift to all entries in the vector then store
  for use in multiplication */
 for(i = 0; i < MAX_SIZE; i++)
  sv[i] = v[i] << vshift;

}/*END OF vshift*/


/*get_vmm_shift will return the amount of shift required for the
  last argument of the APL_VMM command based on the number of
  bits used in the matrix and vector to be supplied to the command */
int get_vmm_shift(int mbits, int vbits)
{
 int mshift, vshift, vmm_shift;

 if(vbits < 1){
  fprintf(stderr, "vector must be composed of at least 1 bit in get_vmm_shift!!\n");
  exit(1);
 }
 if(mbits < 1){
  fprintf(stderr, "matrix must be composed of at least 1 bit in get_vmm_shift!!\n");
  exit(1);
 }
 if((mbits + vbits) > 7){
  fprintf(stderr, "the sum of the number of bits in the matrix and vector can not\n");
  fprintf(stderr, "\tbe greater than 7 for get_vmm_shift!!\n");
  exit(1);
 }

 /*determine the shifts to apply */
 mshift = 8 - mbits;
 vshift = 8 - vbits;

 /*following two conditionals added to avoid sign
   bit causing problems in multiplication */
 if(mshift == 7)
```

```
  mshift--;
 if(vshift == 7)
  vshift--;
 vmm_shift = 15 - (mshift + vshift);

 return vmm_shift;
}/*END OF get_vmm_shift*/


/*HP_MUL will multiply the passed in matrix by the passed in vector.
  The matrix, vector and vector mask are passed in as EnLight256 register
  numbers, and the mask is used to partition the matrix to ensure no
  precision is lost. Note that while m, v and mask_reg must point to
  initialized vectors and matrix, the sum_reg argument only tells
  the command what register to use for summing the partial results
  internally and need no be initialized. mbits and vbits represent
  the maximum number of bits in the matrix and vector respectively,
  vmm_shift is the shift passed to the APL_VMM command */
short int *HP_MUL(int m, int v, int mask_reg, int sum_reg, int mbits,\
                  int vbits, int vmm_shift)
{
 int i, k;
 int max_ent_per_part, num_partitions;
 short int *v_sum;

 if((m < 0)||(v < 0)||(mask_reg < 0)||(sum_reg < 0)||(m > 3)||(v > 15)\
    ||(mask_reg > 15)||(sum_reg > 7)){
  fprintf(stderr, "Invalid register supplied to HP_MUL. Exiting...\n");
  exit(1);
 }
 if((mbits < 1)||(vbits < 1)){
  fprintf(stderr, "matrix and vector in HP_MUL must be composed of at \
                   least 1 bit entries!!\n");
  exit(1);
 }
 if((vmm_shift < 0)||(vmm_shift > 6)){
  fprintf(stderr, "Invalid shift supplied to HP_MUL!! Must be between 0\
                   and 6, got %d!\n", vmm_shift);
  exit(1);
 }
 i = sum_reg * 2;
 k = i++;
 if((v == mask_reg)||(v == i)||(v==k)||(mask_reg == i)||(mask_reg == k)){
  fprintf(stderr, "Invalid register combination passed to HP_MUL!\n");
  fprintf(stderr, "v = %d, mask_reg = %d, sum_reg = %d = %d and %d short\
                   registers used\n.", v, mask_reg, sum_reg, i, k);
  exit(1);
 }
 if((v == 12)||(v == 13)||(v==9)||(mask_reg == 13)||(mask_reg == 12)||\
    (sum_reg==5)||(sum_reg == 6)){
  fprintf(stderr, "Short Vector Registers 12 and 13 are used internally\
                   in HP_MUL\n");
  fprintf(stderr, "Make sure v, mask_reg and sum_reg do not use these\
                    registers.\n ");
  fprintf(stderr, "Also, only mask_reg can be register 9 as it is used internally. \
```

```
                    Exiting...\n");
  exit(1);
 }

 v_sum = (short int*)malloc(sizeof(short int)*MAX_SIZE);
 malloc_error(v_sum, "No memory for v_sum in HP_MUL\n");

 if(mask_reg != 9)
  APL_COPY(mask_reg, 9);

 /*determine the number of partitions to break matrix into */
 max_ent_per_part = (1<<(9-(mbits + vbits))) - 1;
 num_partitions = (MAX_SIZE/max_ent_per_part) + 1;

 /*zero out sum vector for safety */
 for(i = 0; i < MAX_SIZE; i++){
  v_sum[i] = 0;
 }
 /*initialize the internal result vector to all zero */
 DVSET(v_sum, sum_reg);

 /*multiply the matrix by the vector with the approprite mask applied
 and add the resultant vector to the total result */
 for(i = 0; i < num_partitions; i++){

  /*AND the mask with the vector and place the result into VMM vector */
  APL_AND(v, 9, 13);
  APL_VMM(m, 13, 12, vmm_shift); /*M, v, Mv, shift-left */

  /*sum the partial result internally */
  APL_VADDM(sum_reg, 12, sum_reg);

  /*shift the vector mask to multiply the next partition*/
  APL_SHFT_U(9,9);

 }
 /*store the internally summed answer */
 DVEC(sum_reg, v_sum);

 return v_sum;

}/*END OF HP_MUL*/


/*pre_decompm will determine number of matrices
  required for the decomposition and store the number
  required in mb. mbits is the number of bits used in the original
  m, dmbits is the desired number of bits
  for the decomposed matrices and vectors. Note the case where
  dmbits = 1 is only useful in cases where the sign
  is not used */
void pre_decompm(int mbits, int dmbits, int *mb)
{
 if((dmbits < 1)||(dmbits > 5)){
  fprintf(stderr, "The number of bits used to decompose the matrix must be at\n");
```

```
  fprintf(stderr, "least 2 and no greater than 5, and the sum of the bits used\
                   for the\n");
  fprintf(stderr, "matrix and vector must be less than 8 in function\
                   pre_decompm. Exiting...\n");
  exit(1);
 }
 if(mbits < 0){
  fprintf(stderr, "Matrix in pre_decompm must be made of entries\n");
  fprintf(stderr, "\tof at least 1-bit! Exiting...\n");
  exit(1);
 }
 if(mb == NULL){
  fprintf(stderr, "Argument mb in pre_decompm is invalid. Exiting...\n");
  exit(1);
 }

 if(dmbits > 1)
  dmbits--;
 *mb = mbits/dmbits;
 if(mbits%dmbits)
  (*mb)++;
}/*ENF OF pre_decompm*/

/*pre_decompv will determine number of vectors
  required for the decomposition and store the number
  required in vb. vbits is the number of bits used in the original
  v, dvbits is the desired number of bits
  for the decomposed vectors. Note the case where
  dvbits = 1 is only useful in cases where the sign
  is not used*/
void pre_decompv(int vbits, int dvbits, int *vb)
{
 if((dvbits < 1)||(dvbits > 5)){
  fprintf(stderr, "The number of bits used to decompose the vector must be at\n");
  fprintf(stderr, "least 2 and no greater than 5, and the sum of the bits\
                   used for the\n");
  fprintf(stderr, "matrix and vector must be less than 8 in function\
                   pre_decompv. Exiting...\n");
  exit(1);
 }
 if(vbits < 0){
  fprintf(stderr, "vector in pre_decompv must be made of entries\n");
  fprintf(stderr, "\tof at least 1-bit! Exiting...\n");
  exit(1);
 }
 if(vb == NULL){
  fprintf(stderr, "Argument vb in pre_decompv is invalid. Exiting...\n");
  exit(1);
 }

 if(dvbits > 1)
  dvbits--;
 *vb = vbits/dvbits;
 if(vbits%dvbits)
  (*vb)++;
```

```
}/*ENF OF pre_decompv*/


/*decomp_m will decompose the matrix into groups small enough for
  the HP_MUL command to handle, m is the original matrix and
  mdec is a pointer to an array of matrix_type matrices to place the
  decompsed groups of m into. mbits is the max number of bits used in
  m, dmbits the number of bits to create each group from. mb is
  the number of matrix_type matrices to decompose m into */
void decomp_m(long **m, matrix_type *mdec, int dmbits, int mb, int ents)
{
 int i, j, k, t, tn = 0, tt, mask;

 if((dmbits < 1)||(dmbits > 5)){
  fprintf(stderr, "The number of bits used to decompose the matrix be\
                   at least 1 and\n");
  fprintf(stderr, "no greater than 5, and the sum of the bits used for\
                   decomposing the\n");
  fprintf(stderr, "matrix and vector must be less than 8 in function\
                   decomp_m. Exiting...\n");
  exit(1);
 }
 if(m == NULL){
  fprintf(stderr, "m in decomp_m not initialized. Exiting...\n");
  exit(1);
 }
 if(mdec == NULL){
  fprintf(stderr, "mdec in decomp_m not initialized. Exiting...\n");
  exit(1);
 }
 if(mb < 1){
  fprintf(stderr, "Decompositions must be of at least 1 group in decomp_m!\n");
  fprintf(stderr, " matrix groups = %d Exiting...\n", mb);
  exit(1);
 }
 if((ents < 1)||(ents > MAX_SIZE)){
  fprintf(stderr, "Number of entries must be > 0 and < 256 in decomp_m!\
                   Exiting...\n");
  exit(1);
 }


 /*create a mask to extract bits */
 if(dmbits > 1)
  mask = (1 << (dmbits-1))-1;
 else
  mask = 1;

 /*for all m[i][j] break up each entry into mb groups
   of dmbits */
 for(i = 0; i < ents; i++){
  for(j = 0; j < ents; j++){
   t = m[i][j];
   tn = 0;
   /*if entry is negative make it positive while you break it up
```

```
    to ensure 2's complement doesnt mess with the final sums */
   if(t < 0){
t *= -1;
tn = 1;
   }

   for(k = 0; k < mb; k++){
    tt = t&mask;
if(tn)
 tt *= -1;

    mdec[k][i][j] = tt;
    t = t >> (((dmbits > 1)? dmbits : 2) - 1);
   }

  }
 }

}/*END OF decomp_m*/


/*decomp_v will decompose the vector into groups small enough for
  the HP_MUL command to handle, v is the original vector, vdec is
  a pointer to arrays to place the decompsed groups of v into.
  vbits is the number of bits in v, dvbits is the number of bits
  to create each group from. vb is the number of vec_type vectors
  to decompose v into */
void decomp_v(long *v, vec_type *vdec, int dvbits, int vb, int ents)
{
 int i, k, t, tn = 0, tt, mask;

 if((dvbits < 1)||(dvbits > 5)){
  fprintf(stderr, "The number of bits used to decompose the vector must \
                   be at least 1 and\n");
  fprintf(stderr, "no greater than 5, and the sum of the bits used \
                   for decomposing the\n");
  fprintf(stderr, "matrix and vector must be less than 8 in function \
                   decomp_v. Exiting...\n");
  exit(1);
 }
 if(v == NULL){
  fprintf(stderr, "v in decomp_v not initialized. Exiting...\n");
  exit(1);
 }
 if(vdec == NULL){
  fprintf(stderr, "vdec in decomp_v not initialized. Exiting...\n");
  exit(1);
 }
 if(vb < 1){
  fprintf(stderr, "Decompositions must be of at least 1 group in decomp_v!\n");
  fprintf(stderr, " vector groups = %d Exiting...\n", vb);
  exit(1);
 }
 if((ents < 1)||(ents > MAX_SIZE)){
  fprintf(stderr, "Number of entries must be > 0 and < 256 in decomp_v! \
```

```
                  Exiting...\n");
 exit(1);
}

/*create a mask to extract bits */
if(dvbits > 1)
 mask = (1 << (dvbits-1))-1;
else
 mask = 1;

/*for all v[i] break up each entry into vb groups
  of dvbits */
for(i = 0; i < ents; i++){
 t = v[i];
 tn = 0;
 /*if entry is negative make it positive while you break it up
to ensure 2's complement doesnt mess with the final sums */
 if(t < 0){
  t *= -1;
  tn = 1;
 }

 for(k = 0; k < vb; k++){
  tt = t&mask;
  if(tn)
tt *= -1;

  vdec[k][i] = tt;
  t = t >> (((dvbits > 1)? dvbits : 2) - 1);
 }
}

}/*END OF decomp_v*/


/*This function will run through the passed in matrix and find
  the largest absolute number, determine the number of bits it
  uses and return that number + 1*/
int get_mbits(long **m, int r, int c)
{
 int i, j;
 long max = 0, t;

 if(m == NULL){
  fprintf(stderr, "m in get_mbits not initialized. Exiting...\n");
  exit(1);
 }
 if((r < 1)||(r > MAX_SIZE)||(c < 1)||(c > MAX_SIZE)){
  fprintf(stderr, "Number of rows and cols must be > 0 and < 256 in\
                   get_mbits! Exiting...\n");
  exit(1);
 }


 for(i = 0; i < r; i++){
```

```
  for(j = 0; j < c; j++){
   t = m[i][j];
   if(t < 0)
    t *= -1;
   if(t > max)
    max = t;
  }
 }

 /*determine # of bits used in max */
 i = 1;
 while(max){
  max = max >> 1;
  i++;
 }

 return i;
}/*END OF get_mbits*/


/*This function will run through the passed in matrix and find
  the largest absolute number, determine the number of bits it
  uses and return that number + 1*/
int get_vbits(long *v, int r)
{
 int i;
 long max = 0, t;

 if(v == NULL){
  fprintf(stderr, "v in get_vbits not initialized. Exiting...\n");
  exit(1);
 }
 if((r < 1)||(r > MAX_SIZE)){
  fprintf(stderr, "Number of rows must be > 0 and < 256 in get_v_bits!\
                   Exiting...\n");
  exit(1);
 }


 for(i = 0; i < r; i++){
  t = v[i];
  if(t < 0)
   t *= -1;
  if(t > max)
   max = t;
 }

 /*determine # of bits used in max */
 i = 1;
 while(max){
  max = max >> 1;
  i++;
 }

 return i;
```

```
}/*END OF get_vbits*/


/*MUL_56 will multiply a NxN matrix by a N-entry vector on the EnLight256
  hardware given that the sum of the number of bits used by the
  largest entry in the matrix plus number of bits used by the
  largest entry in the vector is not greater than 56. m is
  the matrix to multiply, v is the vector. ans is a
  N-entry vector of type long long*/
void MUL_56(long **m, long *v, long long *ans, int mbits, int vbits, \
    int dmbits, int dvbits, int num_entries)
{
 int vmm_shift, vb, mb, i, j, k;
 short int *v_sum;
 vec_type v_mask, shifted_v, *vdec;
 matrix_type shifted_m, *mdec;

 if(m == NULL){
  fprintf(stderr, "m in MUL_56 not initialized. Exiting...\n");
  exit(1);
 }
 if(v == NULL){
  fprintf(stderr, "v in MUL_56 not initialized. Exiting...\n");
  exit(1);
 }
 if(ans == NULL){
  fprintf(stderr, "ans in MUL_56 not initialized. Exiting...\n");
  exit(1);
 }
 if((num_entries < 1)||(num_entries > MAX_SIZE)){
  fprintf(stderr, "Number of entries must be > 0 and < 256 in MUL_56!\
                   Exiting...\n");
  exit(1);
 }
 if((dvbits < 1)||(dvbits > 5)){
  fprintf(stderr, "The number of bits used to decompose the\
                    vector must be at least 1 and\n");
  fprintf(stderr, "no greater than 5, and the sum of the bits\
                    used for decomposing the\n");
  fprintf(stderr, "matrix and vector must be less than 8 in\
                    function 56. Exiting...\n");
  exit(1);
 }
 if((dmbits < 1)||(dmbits > 5)||((dmbits+dvbits) > 7)){
  fprintf(stderr, "The number of bits used to decompose the\
                    matrix must be at least 1 and\n");
  fprintf(stderr, "no greater than 5, and the sum of the bits\
                    used for decomposing the\n");
  fprintf(stderr, "matrix and vector must be less than 8 in\
                    function MUL_56. Exiting...\n");
  exit(1);
 }
 if((mbits < 1)||(vbits < 1)){
  fprintf(stderr, "Matrix and vector must be made of at least\
                    1-bit entries in MUL_56!\n");
```

```
 exit(1);
}
if((mbits + vbits) > 56){
 fprintf(stderr, "Total of bits used for matrix and vector must\
                  be <= 56 in MUL_56!\n");
 exit(1);
}

/*create vector mask outside of main loop since it can be reused*/
gen_vmask(dmbits, dvbits, v_mask);

/*determine number of matrices and vectors required\
   for the decomposition*/
pre_decompm(mbits, dmbits, &mb);
pre_decompv(vbits, dvbits, &vb);

/*initialize mdec and vdec*/
vdec = (vec_type *) malloc(sizeof(vec_type)*vb);
malloc_error(vdec, "No memory for vdec in MUL_56. Exiting...\n");

mdec = (matrix_type *) malloc(sizeof(matrix_type)*mb);
malloc_error(mdec, "No memory for mdec in MUL_56. Exiting...\n");

/*decompose the matrix and vector into groups small enough for
  the HP_MUL command to handle*/
decomp_m(m, mdec, dmbits, mb, num_entries);
decomp_v(v, vdec, dvbits, vb, num_entries);

vmm_shift = get_vmm_shift(dmbits, dvbits);

/*multiply the parts of the matrix by the parts of the vector */
for(i = 0; i < mb; i++){
 /*shift and store the matrix */
 mshift(dmbits, mdec[i], shifted_m);
 SMSET(shifted_m, M_REG);

 for(j = 0; j < vb; j++){
  /*(re)set the vector partition mask */
  SVSET(v_mask, MASK_REG);

  /*shift and store the vector */
  vshift(dvbits, vdec[j], shifted_v);
  SVSET(shifted_v, V_REG);

  /*perform the multiply*/
  v_sum = HP_MUL(M_REG, V_REG, MASK_REG, SUM_REG, dmbits, dvbits, vmm_shift);

  /*apply the proper shift and add the partial sum to the total vector sum*/
  for(k = 0; k < num_entries; k++){
ans[k] += (long long) v_sum[k] << ((i*(dmbits-1))+(j*(dvbits-1)));
  }
  free(v_sum);
 }
}
```

```
 free(mdec);
 free(vdec);

}/*END OF MUL_56*/


int main(int argc, char *argv[])
{
 int i, j, h, mbits, vbits, dmbits, dvbits, num_entries;
 long **m, *v;
 long long full_sum[MAX_SIZE], k;
 Ulong seed;

 /*read & error check arguments*/
 if(argc != 5){
  usage();
 }
 sscanf(argv[1], "%d", &mbits);
 if((mbits < 1)||(mbits > 31)){
  usage();
 }
 sscanf(argv[2], "%d", &vbits);
 if(((vbits < 1)||(vbits > 31))){
  usage();
 }
 sscanf(argv[3], "%d", &num_entries);
 if((num_entries < 1)||(num_entries > MAX_SIZE)){
  usage();
 }
 sscanf(argv[4], "%lu", &seed);
 if(((mbits + vbits) > 56)){
  usage();
 }

 /*initialize random number generator*/
 initrand(seed);

 /*malloc your matrix and vector*/
 v = (long*) malloc(sizeof(long)*MAX_SIZE);
 malloc_error(v, "v in main");
 m = (long**) malloc(sizeof(long*)*MAX_SIZE);
 malloc_error(m, "m in main");
 for(i= 0; i < MAX_SIZE; i++){
  m[i]= (long*)malloc(sizeof(long)*MAX_SIZE);
  malloc_error(m, "m in main");
  v[i] = 0;
  for(j = 0; j < MAX_SIZE; j++)
   m[i][j] = 0;
 }

 /*note you can set dvbits or dmbits to 1 iff
   v or m respectively only have unsigned entries,
   ie 1-bit entries*/
 dmbits = 5; dvbits = 2;
```

```
 for(i=0; i < MAX_SIZE; i++)
full_sum[i] = 0;

 /*generate a nice matrix and vectori */
 gen_mv(m, v, mbits, vbits, num_entries);

 /*call the multiply function*/
 MUL_56(m, v, full_sum, mbits, vbits, dmbits, dvbits, num_entries);

 /*compare real answer to enlight answer*/
 for(i = 0; i < num_entries; i++){
  k = 0;
  for(j = 0; j < num_entries; j++){
   k += (long long) m[i][j]*v[j];
  }
  if(full_sum[i] != k){
printf("full_sum[%d] %lld != %lld (EL256)!", i, full_sum[i], k);
    print_vec(num_entries, "VECTOR!: ", v);
printf("MATRIX ROW:\n");
for(h = 0; h < num_entries; h++){
      printf("%ld ", m[i][h]);
}
    printf("\nREAL ANSWER AND ENLIGHT ANSWER DO NOT MATCH!!\n");
    exit(0);
   }
 }


 printf("\nREAL ANSWER == ENLIGHT ANSWER!! AWWW YEAAHHH!\n");

 return 0;
}/*END OF main*/


/*
gcc -I .-Wall -gdwarf-2 -g3 -o 56bitmm 56bitmm.c -lm
gcc -I .-Wall -O3 -mfpmath=387 -o 56bitmm 56bitmm.c -lm
*/
```

Appendix F

```
/*
Jacobian method for solving Ax = b
A is a symmetric matrix (dimension num_entries),  and x, b are vectors.
The method is iterative and will run until either some number of iterations has occured
or there is convergence.
*/
#include "enlight256_hp.h"
#include "random32.h"

double      B[MAX_SIZE], D[MAX_SIZE], Xk[MAX_SIZE], A[MAX_SIZE][MAX_SIZE];
long L_U[MAX_SIZE][MAX_SIZE], X[MAX_SIZE];
long long ans[MAX_SIZE];

void printm(long m[MAX_SIZE][MAX_SIZE], int n)
{
  int i,j;

  for (i = 0; i < n; i++){
    for (j = 0; j < n; j++) printf("%ld ",m[i][j]);
    putchar('\n');
  }
}

void gen_m(double m[MAX_SIZE][MAX_SIZE], int n, int mb)
{
  int i,j,k,l;  double t, x = 0, ld;

  l = 1 << mb;
  ld = (double)l/2 - .1;
  for (i = 0; i < MAX_SIZE; i++){
    for (t = 0, k = 0; k < i; k++) t += fabs(m[k][i]);
    for (j = i; j < MAX_SIZE; j++)
      if ((i < n)&&(j < n)&&(i < j)){
        t += fabs(m[j][i] = m[i][j] = U01-0.5);     // sum abs off diagonal
        if (fabs(m[i][j]) > x) x = fabs(m[i][j]);
      } else {
        m[j][i] = m[i][j] = 0.;
      }
    m[i][i] = t;
  }
  for (i = 0; i < n; i++){
    for (t = 0, j = i; j < n; j++)
      if ((i < n)&&(j < n)){
```

```
        m[j][i] = m[i][j] = trunc(ld*m[i][j]/x);
      }
  }
}

void gen_v(double *v, int n, int vb)
{
  int i, l;

  l = 1 << vb;
  for (i = MAX_SIZE; i--; v[i] = (int)(l*((i<n)? U01-0.5: 0.)));
}

double check_convergence(int n, double tolerance)
{
  int i,j;  double t,b,d;

  for (b = d = i = 0; i < n; i++){
    for (t = 0, j = n; j--; t += X[j]*A[i][j]);
    d += (t-B[i])*(t-B[i]);
    b += B[i]*B[i];
  }
  if ((d = sqrt(d/b)) > tolerance) return 0.;
  return d;
}



int main(int argc, char *argv[])
{
  int n,iterations,i,j,k,l,mbits=28,vbits=28;  double t;  vec_type y_k,b;

  l = 1 << mbits;
  if (argc == 4){
    sscanf(argv[1], "%d", &n);           // matrix size
    sscanf(argv[2],"%lf", &t);           // tolerance (relative error)
    sscanf(argv[3], "%d", &iterations);
for(iterations = 1; iterations <= 20; iterations++){
clear_stats(NULL);
    if ((iterations > 0)&&(n > 1)&&(n <= MAX_SIZE)){
      initrand(7);            // seed
      gen_m(A,n,mbits);             // random matrix
      gen_v(Xk,n,vbits);            // random answer

      for (i = 0; i < MAX_SIZE; i++){
        for (B[i] = j = 0; j < MAX_SIZE; j++){
          if (!((i < n)&&(j < n))){
            L_U[i][j] = 0;
            continue;
          }
          B[i] += A[i][j]*Xk[j];        // compute B as the product of A and Xk
          if (i == j){
            D[i] = A[i][j];
            L_U[i][j] = 0;
          }
```

```
          else L_U[i][j] = (long) A[i][j];
        }
        b[i] = (short int)((i < n)? B[i]: 0.);
      }

      for (j = n; j--; X[j] = (signed char)(1*(U01-.5))); // initial guess

      for (k = iterations; k--; ){         // Jacobi method
   MUL_56(L_U, X, ans, mbits, vbits, 4, 3, n);

   for(i=0; i < n; i++){
    X[i] = ((B[i]-ans[i])/D[i]);
ans[i] = 0;
   }

      }

      if ((t = check_convergence(n,t)) > 0.)  // tolerance is relative error
        printf("\nConvergence in %d iterations (%lf).\n", iterations,t);
      else
        printf("\nDid not converge in %d iterations\n", iterations);

      printf("Jacobi gives: \n");
      for (i = 0; i < n; printf("%ld ",X[i++]));
      putchar('\n');
      printf("Answer is: \n");
      for (i = 0; i < n; printf("%.0lf ",Xk[i++]));
      putchar('\n');
  Report(NULL,"main");
    }
  }
  Report(NULL,"main");
  }
  return 0;
}

/*
gcc -Wall -O3 -o jacobi_enlight jacobi_enlight.c -lm  $(INCLUDE)/modded_sim
$(INCLUDE)/random_gen $(INCLUDE)/enlight256_hp
*/
```

Appendix G

```
/*enlight256_hp.h:      ADAM GRAHAM, FALL '05
                        UNIVERSITY OF TENNESSEE,
                        KNOXVILLE
  This library allows high precision with the 8-bit
  EnLight256.*/

#include "simulate.c" /*Include this line only if running off of
                                                    the EnLight256 simulator*/

/*declare the max matrix/vector size and the registers you'd like
  to use on the EnLight256. Note that the SUM_REG refers to a
  long vector register (thus takes up two concurrent short vector
  registers), M_REG to a matrix register, and MASK_REG and
  V_REG are short vector registers*/
#define MAX_SIZE 256
#define MASK_REG 6
#define M_REG 0
#define V_REG 0
#define SUM_REG 2

typedef signed char vec_type[MAX_SIZE];

/*This checks to see if malloc failed and
    prints out an error message if ti does. */
void *Malloc(int n, char *s)
{
  void *p = malloc(n);

  if (p) return p;
  fprintf(stderr, "Memory Allocation error in %s\n Exiting...", s);
  exit(1);
}

void isNULL(void *p, char *s)
{
  if (p != NULL) return;
  fprintf(stderr, "%s Exiting...\n",s);
  exit(1);
}

/*this is just to print data in vectors to screen*/
void print_vec(int num_els, char *message,  long *print_me)
{
```

```
  int i;

  if(message)
    printf("\n %s\n", message);

  for(i = 0; i < num_els; i++){
    if((i>0)&&(!(i%16)))
      printf("\n");
    printf("%8ld ", print_me[i]);
  }
  printf("\n");
}


/*test_m tests to make sure that the decomposition m of was done
  correctly and no data loss has occurred mb = number of groups,
  m is the original square matrix, dmbits is the number of bits used in
  decomposing the matrix*/
void test_m(long **m, int dmbits, int mb, matrix_type *mdec, int num_entries)
{
  int i, j, k, h;

  for(i = 0; i < num_entries; i++){
    for(j = 0; j < num_entries; j++){
      for(h=0, k=0; k < mb; k++){
h+= mdec[k][i][j] << (k*(dmbits-1));
      }
      if(h != m[i][j]){
printf("m[%d] %ld!= %d\n", i, m[i][j], h);
exit(1);
      }
    }
  }
  printf("M OK!!\n");
}


/*test_v tests to make sure that the decomposition of v was done
  correctly and no data loss has occurred. vb = number of groups,
  v is original vector, dvbits is the  number of bits used in
  decomposing the vector */
void test_v(long *v, int dvbits, int vb, vec_type *vdec, int num_entries)
{
  int i, j, h;

  for(i = 0; i < num_entries; i++){
    for(h=0, j=0; j < vb; j++){
      h+= vdec[j][i] << (j*(dvbits-1));
    }
    if(h != v[i]){
      printf("v[%d] %ld!= %d\n", i, v[i], h);
      exit(1);
    }
  }
  printf("V OK!!\n");
```

```
}


/*gen_vmask will generate the vector mask applied to the
  vector to be multiplied by the matrix in HP_MUL.*/
void gen_vmask(int mbits, int vbits, vec_type v_mask)
{
  int max_ent, num_partitions, i;

  isNULL(v_mask,"v_mask in gen_vmask not initialized.");

  if((mbits < 1)||(vbits < 1)){
    fprintf(stderr, "matrix and vector must be composed of at least 1 bit entries!!\n");
    exit(1);
  }

  /*determine the number of partitions to break matrix into*/
  max_ent = (1<<(9-(mbits + vbits))) - 1;
  num_partitions = ceil(((double)MAX_SIZE)/max_ent);

  /*create the vector partition mask*/
  for(i = 0; i < MAX_SIZE; i++)
    v_mask[i] = ((i%num_partitions)? 0: -1);
}


/*mshift will determine, and return the shift amount, and apply the
  appropriate shift to the matrix m and store the shifted results in sm*/
int mshift(int mbits, matrix_type m, matrix_type sm )
{
  int mshift, i, j;

  isNULL(m ,"m  in mshift not initialized.");
  isNULL(sm,"sm in mshift not initialized.");

  if(mbits < 1){
    fprintf(stderr, "matrix must be composed of at least 1 bit entries in mshift!!\n");
    exit(1);
  }

  /*determine the shifts to apply*/
  mshift = 8 - mbits;

  /*following condition added to avoid sign
    bit causing problems in multiplication */
  if(mshift == 7)
    mshift--;

  /*apply max shift to all entries in matrix then store
    for use in multiplication */
  for(i = 0; i < MAX_SIZE; i++){
    for(j = 0; j < MAX_SIZE; j++){
      sm[i][j] = m[i][j] << mshift;
    }
  }
```

```
  return mshift;
}


/*vshift will determine and apply the appropriate shift to the vector v
  and store the shifted results in sv */
int vshift(int vbits, vec_type v, vec_type sv)
{
  int vshift, i;

  isNULL(v,"v in vshift not initialized.");
  isNULL(sv,"sv in vshift not initialized.");

  if(vbits < 1){
    fprintf(stderr, "vector must be composed of at least 1 bit entries in vshift!!\n");
    exit(1);
  }

  /*determine the shifts to apply */
  vshift = 8 - vbits;

  /*following condition added to
    avoid sign bit causing problems in multiplication */
  if(vshift == 7)
    vshift--;

  /*apply max shift to all entries in the vector then store
    for use in multiplication */
  for(i = 0; i < MAX_SIZE; i++)
    sv[i] = v[i] << vshift;

  return vshift;
}


/*get_vmm_shift will return the amount of shift required for the
  last argument of the APL_VMM command based on the number of
  bits used in the matrix and vector to be supplied to the command */
int get_vmm_shift(int mbits, int vbits)
{
  int mshift, vshift, vmm_shift;

  if(vbits < 1){
    fprintf(stderr, "vector must be composed of at least 1 bit in get_vmm_shift!!\n");
    exit(1);
  }
  if(mbits < 1){
    fprintf(stderr, "matrix must be composed of at least 1 bit in get_vmm_shift!!\n");
    exit(1);
  }
  if((mbits + vbits) > 7){
    fprintf(stderr, "the sum of the number of bits in the matrix and vector can not\n");
    fprintf(stderr, "\tbe greater than 7 for get_vmm_shift!!\n");
    exit(1);
  }
```

```
  /*determine the shifts to apply */
  mshift = 8 - mbits; vshift = 8 - vbits;

  /*following two conditionals added to avoid sign
    bit causing problems in multiplication */
  if(mshift == 7)
    mshift--;
  if(vshift == 7)
    vshift--;
  vmm_shift = 15 - (mshift + vshift);
  return vmm_shift;
}


/*HP_MUL will multiply the passed in matrix by the passed in vector.
  The matrix, vector and vector mask are passed in as EnLight256 register numbers,
  and the mask is used to partition the matrix to ensure no precision is lost.
  Note that while m, v and mask_reg must point to initialized vectors and matrix,
  the sum_reg argument only tells the command what register to use for summing
  the partial results internally and need no be initialized. mbits and
  vbits represent the maximum number of bits in the matrix and vector
  respectively, vmm_shift is the shift passed to the APL_VMM command. */
void HP_MUL(int m, int v, int mask_reg, int sum_reg, int mbits, int vbits, int vmm_shift, \
short int *v_sum)
{
  int i, k;
  int max_ent_per_part, num_partitions;

  if((m < 0)||(v < 0)||(mask_reg < 0)||(sum_reg < 0)||(m > 3)||(v > 15)||(mask_reg > 15) \
            ||(sum_reg > 7)){
    fprintf(stderr, "Invalid register supplied to HP_MUL. Exiting...\n");
    exit(1);
  }
  if((mbits < 1)||(vbits < 1)){
    fprintf(stderr, "matrix and vector in HP_MUL must be composed of at least 1 bit \
           entries!!\n");
    exit(1);
  }
  if((vmm_shift < 0)||(vmm_shift > 6)){
    fprintf(stderr, "Invalid shift supplied to HP_MUL!! Must be between 0 and 6, got %d!\n"\
     , vmm_shift);
    exit(1);
  }
  i = sum_reg * 2;
  k = i++;
  if((v == mask_reg)||(v == i)||(v==k)||(mask_reg == i)||(mask_reg == k)){
    fprintf(stderr, "Invalid register combination passed to HP_MUL!\n");
    fprintf(stderr, "v = %d, mask_reg = %d, sum_reg = %d = %d and %d short registers used\n.",\
      v, mask_reg, sum_reg, i, k);
    exit(1);
  }
  if((v == 12)||(v == 13)||(v==9)||(mask_reg == 13)||(mask_reg == 12)||(sum_reg==5)||\
   (sum_reg == 6)){
    fprintf(stderr, "Short Vector Registers 12 and 13 are used internally in HP_MUL\n");
```

```
    fprintf(stderr, "Make sure v, mask_reg and sum_reg do not use these registers.\n ");
    fprintf(stderr, "Also, only mask_reg can be register 9 as it is used internally.\
      Exiting...\n");
    exit(1);
  }

  if(mask_reg != 9)
    APL_COPY(mask_reg, 9);

  /*determine the number of partitions to break matrix into */
  max_ent_per_part = (1<<(9-(mbits + vbits))) - 1;
  num_partitions = ceil(((double)MAX_SIZE)/max_ent_per_part);

  /*initialize the internal result vector to all zero */
  APL_SMUL(0,0,sum_reg);

  /*multiply the matrix by the vector with the approprite mask applied
    and add the resultant vector to the total result */
  i = 0;

  /*AND the mask with the vector and place the result into VMM vector */
  APL_AND(v, 9, 13);
  APL_VMM(m, 13, 12, vmm_shift); /*M, v, Mv, shift-left */

  /*sum the partial result internally */
  APL_VADDM(sum_reg, 12, sum_reg);

  while(++i < num_partitions){
    /*shift the vector mask to multiply the next partition*/
    APL_SHFT_U(9,9);

    /*AND the mask with the vector and place the result into VMM vector */
    APL_AND(v, 9, 13);
    APL_VMM(m, 13, 12, vmm_shift); /*M, v, Mv, shift-left */

    /*sum the partial result internally */
    APL_VADDM(sum_reg, 12, sum_reg);
  }

  /*store the internally summed answer */
  DVEC(sum_reg, v_sum);
}


/*pre_decompm will determine number of matrices
  required for the decomposition and store the number
  required in mb. mbits is the number of bits used in the original
  m, dmbits is the desired number of bits
  for the decomposed matrices and vectors. Note the case where
  dmbits = 1 is only useful in cases where the sign
  is not used */
void pre_decompm(int mbits, int dmbits, int *mb)
{
  if((dmbits < 1)||(dmbits > 5)){
    fprintf(stderr, "The number of bits used to decompose the matrix must be at\n");
```

```
    fprintf(stderr, "least 2 and no greater than 5, and the sum of the bits used for the\n");
    fprintf(stderr, "matrix and vector must be less than 8 in function pre_decompm.\
      Exiting...\n");
    exit(1);
  }
  if(mbits < 0){
    fprintf(stderr, "Matrix in pre_decompm must be made of entries\n");
    fprintf(stderr, "\tof at least 1-bit! Exiting...\n");
    exit(1);
  }
  isNULL(mb,"Argument mb in pre_decompm is invalid.");

  if(dmbits > 1)
    dmbits--;
  *mb = ceil(((double)mbits)/dmbits);
}


/*pre_decompv will determine number of vectors
  required for the decomposition and store the number
  required in vb. vbits is the number of bits used in the original
  v, dvbits is the desired number of bits
  for the decomposed vectors. Note the case where
  dvbits = 1 is only useful in cases where the sign
  is not used*/
void pre_decompv(int vbits, int dvbits, int *vb)
{
  if((dvbits < 1)||(dvbits > 5)){
    fprintf(stderr, "The number of bits used to decompose the vector must be at\n");
    fprintf(stderr, "least 2 and no greater than 5, and the sum of the bits used for the\n");
    fprintf(stderr, "matrix and vector must be less than 8 in function pre_decompv.\
      Exiting...\n");
    exit(1);
  }
  if(vbits < 0){
    fprintf(stderr, "vector in pre_decompv must be made of entries\n");
    fprintf(stderr, "\tof at least 1-bit! Exiting...\n");
    exit(1);
  }
  isNULL(vb,"Argument vb in pre_decompv is invalid.");

  if(dvbits > 1)
    dvbits--;
  *vb = ceil(((double)vbits)/dvbits);
}


/*decomp_m will decompose the matrix into groups small enough for
  the HP_MUL command to handle, m is the original matrix and
  mdec is a pointer to an array of matrix_type matrices to place the
  decompsed groups of m into. mbits is the max number of bits used in
  m, dmbits the number of bits to create each group from. mb is
  the number of matrix_type matrices to decompose m into */
void decomp_m(long **m, matrix_type *mdec, int dmbits, int mb, int ents)
{
```

```
  int i, j, k, t, tn = 0, mask;

  if((dmbits < 1)||(dmbits > 5)){
    fprintf(stderr, "The number of bits used to decompose the matrix be at least 1 and\n");
    fprintf(stderr, "no greater than 5, and the sum of the bits used for decomposing the\n");
    fprintf(stderr, "matrix and vector must be less than 8 in function decomp_m.\
      Exiting...\n");
    exit(1);
  }
  isNULL(m,"m in decomp_m not initialized.");
  isNULL(mdec,"mdec in decomp_m not initialized.");
  if(mb < 1){
    fprintf(stderr, "Decompositions must be of at least 1 group in decomp_m!\n");
    fprintf(stderr, " matrix groups = %d Exiting...\n", mb);
    exit(1);
  }
  if((ents < 1)||(ents > MAX_SIZE)){
    fprintf(stderr, "Number of entries must be > 0 and < 256 in decomp_m! Exiting...\n");
    exit(1);
  }

  /*create a mask to extract bits */
  if(dmbits > 1)
    mask = (1 << (dmbits-1))-1;
  else
    mask = 1;

  /*for all m[i][j] break up each entry into mb groups
    of dmbits */
  for(i = 0; i < ents; i++){
    for(j = 0; j < ents; j++){
      t = m[i][j];
      /*if entry is negative make it positive while you break it up
to ensure 2's complement doesnt mess with the final sums */
      if(t < 0){
        t = -t;
        tn = -1;
      } else tn = 1;

      for(k = 0; k < mb; k++){
mdec[k][i][j] = (t&mask)*tn;
t >>= ((dmbits > 1)? dmbits-1: 1);
      }
    }
  }
}


/*decomp_v will decompose the vector into groups small enough for
  the HP_MUL command to handle, v is the original vector, vdec is
  a pointer to arrays to place the decompsed groups of v into.
  vbits is the number of bits in v, dvbits is the number of bits
  to create each group from. vb is the number of vec_type vectors
  to decompose v into */
void decomp_v(long *v, vec_type *vdec, int dvbits, int vb, int ents)
```

```
{
  int i, k, t, tn = 0, mask;

  if((dvbits < 1)||(dvbits > 5)){
    fprintf(stderr, "The number of bits used to decompose the vector must be at least \
     1 and\n");
    fprintf(stderr, "no greater than 5, and the sum of the bits used for decomposing the\n");
    fprintf(stderr, "matrix and vector must be less than 8 in function decomp_v.\
      Exiting...\n");
    exit(1);
  }
  isNULL(v,"v in decomp_v not initialized.");
  isNULL(vdec,"vdec in decomp_v not initialized.");
  if(vb < 1){
    fprintf(stderr, "Decompositions must be of at least 1 group in decomp_v!\n");
    fprintf(stderr, " vector groups = %d Exiting...\n", vb);
    exit(1);
  }
  if((ents < 1)||(ents > MAX_SIZE)){
    fprintf(stderr, "Number of entries must be > 0 and < 256 in decomp_v! Exiting...\n");
    exit(1);
  }

  /*create a mask to extract bits */
  if(dvbits > 1)
    mask = (1 << (dvbits-1))-1;
  else
    mask = 1;

  /*for all v[i] break up each entry into vb groups
    of dvbits */
  for(i = 0; i < ents; i++){
    t = v[i];
    /*if entry is negative make it positive while you break it up
      to ensure 2's complement doesnt mess with the final sums */
    if(t < 0){
      t = -t;
      tn = -1;
    } else tn = 1;

    for(k = 0; k < vb; k++){
      vdec[k][i] = (t&mask)*tn;
      t >>= ((dvbits > 1)? dvbits-1: 1);
    }
  }
}


/*This function will run through the passed in matrix and find
  the largest absolute number, determine the number of bits it
  uses and return that number + 1*/
int get_mbits(long **m, int r, int c)
{
  int i, j;
  long max = 0, t;
```

```
  isNULL(m,"m in get_mbits not initialized.");
  if((r < 1)||(r > MAX_SIZE)||(c < 1)||(c > MAX_SIZE)){
    fprintf(stderr, "Number of rows and cols must be > 0 and < 256 in get_mbits!\
      Exiting...\n");
    exit(1);
  }

  for(i = 0; i < r; i++){
    for(j = 0; j < c; j++){
      t = m[i][j];
      if(t < 0)
t *= -1;
      if(t > max)
max = t;
    }
  }

  /*determine # of bits used in max */
  i = 1;
  while(max){
    max = max >> 1;
    i++;
  }
  return i;
}


/*This function will run through the passed in matrix and find
  the largest absolute number, determine the number of bits it
  uses and return that number + 1*/
int get_vbits(long *v, int r)
{
  int i;
  long max = 0, t;

  isNULL(v,"v in get_vbits not initialized.");
  if((r < 1)||(r > MAX_SIZE)){
    fprintf(stderr, "Number of rows must be > 0 and < 256 in get_v_bits! Exiting...\n");
    exit(1);
  }

  for(i = 0; i < r; i++){
    t = v[i];
    if(t < 0)
      t *= -1;
    if(t > max)
      max = t;
  }

  /*determine # of bits used in max */
  i = 1;
  while(max){
    max = max >> 1;
    i++;
```

```
  }
  return i;
}


/*MUL_56 will multiply a NxN matrix by a N-entry vector on the EnLight256
  hardware given that the sum of the number of bits used by the
  largest entry in the matrix plus number of bits used by the
  largest entry in the vector is not greater than 56. m is
  the matrix to multiply, v is the vector. ans is a
  N-entry vector of type long long*/
void MUL_56(long **m, long *v, long long *ans, int mbits, int vbits, \
                   int dmbits, int dvbits, int num_entries)
{
  int vmm_shift, vb, mb, i, j, k;
  short int v_sum[MAX_SIZE];
  vec_type v_mask, shifted_v, *vdec;
  matrix_type shifted_m, *mdec;

  isNULL(m,"m in MUL_56 not initialized.");
  isNULL(v,"v in MUL_56 not initialized.");
  isNULL(ans,"ans in MUL_56 not initialized.");
  if((num_entries < 1)||(num_entries > MAX_SIZE)){
    fprintf(stderr, "Number of entries must be > 0 and < 256 in MUL_56! Exiting...\n");
    exit(1);
  }
  if((dvbits < 1)||(dvbits > 5)){
    fprintf(stderr, "The number of bits used to decompose the vector must be at least\
      1 and\n");
    fprintf(stderr, "no greater than 5, and the sum of the bits used for decomposing the\n");
    fprintf(stderr, "matrix and vector must be less than 8 in function 56. Exiting...\n");
    exit(1);
  }
  if((dmbits < 1)||(dmbits > 5)||((dmbits+dvbits) > 7)){
    fprintf(stderr, "The number of bits used to decompose the matrix must be at least \
     1 and\n");
    fprintf(stderr, "no greater than 5, and the sum of the bits used for decomposing the\n");
    fprintf(stderr, "matrix and vector must be less than 8 in function MUL_56. Exiting...\n");
    exit(1);
  }
  if((mbits < 1)||(vbits < 1)){
    fprintf(stderr, "Matrix and vector must be made of at least 1-bit entries in MUL_56!\n");
    exit(1);
  }
  if((mbits + vbits) > 56){
    fprintf(stderr, "Total of bits used for matrix and vector must be <= 56 in MUL_56!\n");
    exit(1);
  }

  /*create vector mask outside of main loop since it can be reused*/
  gen_vmask(dmbits, dvbits, v_mask);

  /*determine number of matrices and vectors required for the decomposition*/
  pre_decompm(mbits, dmbits, &mb);
  pre_decompv(vbits, dvbits, &vb);
```

```
  /*initialize mdec and vdec*/
  vdec = (vec_type *) Malloc(sizeof(vec_type)*vb,"No memory for vdec in MUL_56.\
    Exiting...\n");

  mdec = (matrix_type *) Malloc(sizeof(matrix_type)*mb,"No memory for mdec in MUL_56.\
    Exiting...\n");

  /*decompose the matrix and vector into groups small enough for
    the HP_MUL command to handle*/
  decomp_m(m, mdec, dmbits, mb, num_entries);
  decomp_v(v, vdec, dvbits, vb, num_entries);

  vmm_shift = get_vmm_shift(dmbits, dvbits);

  /*multiply the parts of the matrix by the parts of the vector */
  for(i = 0; i < mb; i++){
    /*shift and store the matrix */
    mshift(dmbits, mdec[i], shifted_m);
    SMSET(shifted_m, M_REG);

    for(j = 0; j < vb; j++){
      /*(re)set the vector partition mask */
      SVSET(v_mask, MASK_REG);

      /*shift and store the vector */
      vshift(dvbits, vdec[j], shifted_v);
      SVSET(shifted_v, V_REG);

      /*perform the multiply*/
      HP_MUL(M_REG, V_REG, MASK_REG, SUM_REG, dmbits, dvbits, vmm_shift, v_sum);

      /*apply the proper shift and add the partial sum to the total vector sum*/
      for(k = 0; k < num_entries; k++){
        ans[k] += (long long) v_sum[k] << ((i*(dmbits-1))+(j*(dvbits-1)));
      }
    }
  }
  free(mdec);
  free(vdec);
}
```

Appendix H

```
/*
Jacobian method for solving Ax = b
A is a symmetric matrix (dimension num_entries),  and x, b are vectors.
The method is iterative and will run until either some number of iterations has occured
or there is convergence.
*/
#include "enlight256_hp.h"
#include "random32.h"

double      B[MAX_SIZE], D[MAX_SIZE], Xk[MAX_SIZE], A[MAX_SIZE][MAX_SIZE];
long L_U[MAX_SIZE][MAX_SIZE], X[MAX_SIZE];
long long ans[MAX_SIZE];

void printm(long m[MAX_SIZE][MAX_SIZE], int n)
{
  int i,j;

  for (i = 0; i < n; i++){
    for (j = 0; j < n; j++) printf("%ld ",m[i][j]);
    putchar('\n');
  }
}

void gen_m(double m[MAX_SIZE][MAX_SIZE], int n, int mb)
{
  int i,j,k,l;  double t, x = 0, ld;

  l = 1 << mb;
  ld = (double)l/2 - .1;
  printf("ld = %lf\n", ld);
  for (i = 0; i < MAX_SIZE; i++){
    for (t = 0, k = 0; k < i; k++) t += fabs(m[k][i]);
    for (j = i; j < MAX_SIZE; j++)
      if ((i < n)&&(j < n)&&(i < j)){
        t += fabs(m[j][i] = m[i][j] = U01-0.5);     // sum abs off diagonal
        if (fabs(m[i][j]) > x) x = fabs(m[i][j]);
      } else {
        m[j][i] = m[i][j] = 0.;
      }
    m[i][i] = t;
  }
  for (i = 0; i < n; i++){
    for (t = 0, j = i; j < n; j++)
```

```
      if ((i < n)&&(j < n)){
        m[j][i] = m[i][j] = trunc(ld*m[i][j]/x);
      }
  }
}

void gen_v(double *v, int n, int vb)
{
  int i, l;

  l = 1 << vb;
  for (i = MAX_SIZE; i--; v[i] = (int)(l*((i<n)? U01-0.5: 0.)));
}

double check_convergence(int n, double tolerance)
{
  int i,j;  double t,b,d;

  for (b = d = i = 0; i < n; i++){
    for (t = 0, j = n; j--; t += X[j]*A[i][j]);
    d += (t-B[i])*(t-B[i]);
    b += B[i]*B[i];
  }
  if ((d = sqrt(d/b)) > tolerance) return 0.;
  return d;
}



int main(int argc, char *argv[])
{
  int n,iterations,i,j,k,l;  double t;  vec_type y_k,b;
  int mb, vb, mbi, vbi, vmm_shift;
  matrix_type shifted_m, *mdec;
  vec_type v_mask, shifted_v, *vdec;
  short int *v_sum;

  l = 1 << 12;
  if (argc == 4){
    sscanf(argv[1], "%d", &n);           // matrix size
    sscanf(argv[2],"%lf", &t);           // tolerance (relative error)
    sscanf(argv[3], "%d", &iterations);
    if ((iterations > 0)&&(n > 1)&&(n <= MAX_SIZE)){
      initrand(7);            // seed
      gen_m(A,n,12);             // random matrix
      gen_v(Xk,n,12);            // random answer

      for (i = 0; i < MAX_SIZE; i++){
        for (B[i] = j = 0; j < MAX_SIZE; j++){
          if (!((i < n)&&(j < n))){
            L_U[i][j] = 0;
            continue;
          }
          B[i] += A[i][j]*Xk[j];        // compute B as the product of A and Xk
          if (i == j){
```

```
            D[i] = A[i][j];
            L_U[i][j] = 0;
          }
          else L_U[i][j] = (long) A[i][j];
        }
        b[i] = (short int)((i < n)? B[i]: 0.);
      }

 /*create vector mask outside of main loop since it can be reused*/
 gen_vmask(4, 3, v_mask);

 /*decompose matrix into groups and load the groups into
   the EnLight256's matrix vector for rapid multiplication*/
 pre_decompm(12, 4, &mb);
 mdec = (matrix_type *) malloc(sizeof(matrix_type)*mb);
 ag_malloc_error(mdec, "No memory for mdec in MUL_56. Exiting...\n");
 decomp_m(L_U, mdec, 4, mb, n);
 for(i = 0; i < mb; i++){
  /*shift and store the matrix */
  mshift(4, mdec[i], shifted_m);
  SMSET(shifted_m, i);
 }

 pre_decompv(15, 3, &vb);
 vdec = (vec_type *) malloc(sizeof(vec_type)*vb);
 ag_malloc_error(vdec, "No memory for vdec in MUL_56. Exiting...\n");

 vmm_shift = get_vmm_shift(4, 3);


      for (j = n; j--; X[j] = (signed char)(l*(U01-.5))); // initial guess

      for (k = iterations; k--; ){         // Jacobi method
   //MUL_56(L_U, X, ans, 12, 12, 4, 3, n);
decomp_v(X, vdec, 3, vb, n);

       for(mbi = 0; mbi < mb; mbi++){
        for(vbi = 0; vbi < vb; vbi++){
     /*(re)set the vector partition mask */
         SVSET(v_mask, MASK_REG);

         /*shift and store the vector */
 vshift(3, vdec[vbi], shifted_v);
         SVSET(shifted_v, V_REG);

         /*perform the multiply*/
         v_sum = HP_MUL(mbi, V_REG, MASK_REG, SUM_REG, 4, 3, vmm_shift);

         /*apply the proper shift and add the partial sum to the total vector sum*/
         for(l = 0; l < n; l++){
      ans[l] += (long long) v_sum[l] << ((mbi*(4-1))+(vbi*(3-1)));
         }
        free(v_sum);
   }
   for(i=0; i < n; i++){
```

```
    X[i] = ((B[i]-ans[i])/D[i]);
ans[i] = 0;
   }

      }
     }
      if ((t = check_convergence(n,t)) > 0.)  // tolerance is relative error
        printf("\nConvergence in %d iterations (%lf).\n", iterations,t);
      else
        printf("\nDid not converge in %d iterations\n", iterations);

      printf("Jacobi gives: \n");
      for (i = 0; i < n; printf("%ld ",X[i++]));
      putchar('\n');
      printf("Answer is: \n");
      for (i = 0; i < n; printf("%.0lf ",Xk[i++]));
      putchar('\n');
    }
  }
  Report(NULL,"main");
  return 0;
}

/*
gcc -Wall -O3 -o jacobi_enlight jacobi_enlight.c -lm  $(INCLUDE)/modded_sim
 $(INCLUDE)/random_gen $(INCLUDE)/enlight256_hp
*/
```

Vita

Born in Lindsay, Ontario on October 7th, 1980, to Bruce and Cynthia Graham, Adam Donald Graham moved to Tennessee with his family in 1996 where he graduated high school at Coffee County Central High School. He then went on to recive a Bachelor of Science degree *cum laude* from the University of Tennessee from the Department of Computer Science. Following this he received his Master of Science degree, also in Computer Science, from the University of Tennessee in the Fall of 2005.